

University of Warwick institutional repository: <http://go.warwick.ac.uk/wrap>

A Thesis Submitted for the Degree of PhD at the University of Warwick

<http://go.warwick.ac.uk/wrap/2314>

This thesis is made available online and is protected by original copyright.

Please scroll down to view the document itself.

Please refer to the repository record for this item for information to help you to cite it. Our policy information is available from the repository home page.

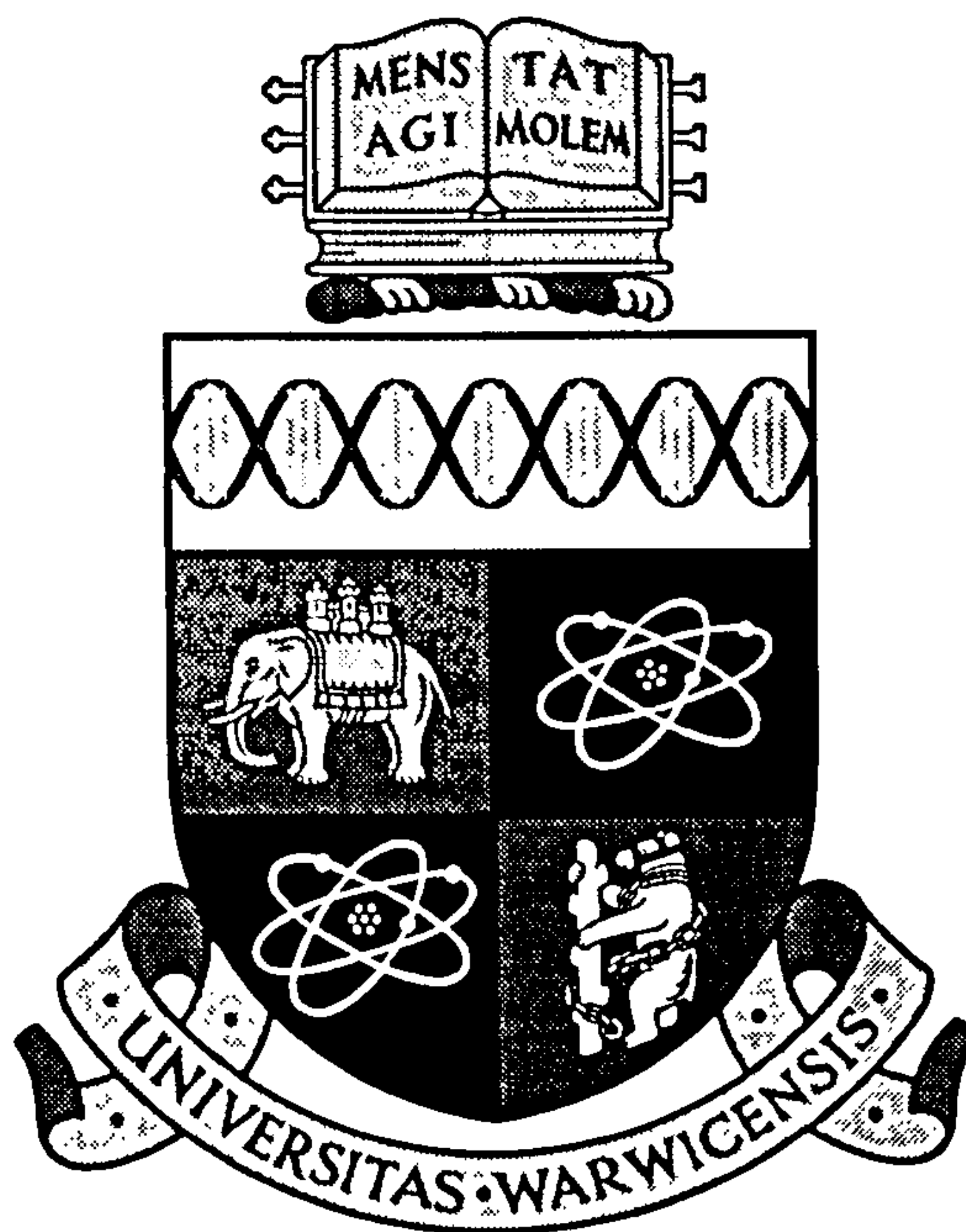
Strategies for producing fast finite element solutions of the incompressible Navier-Stokes equations on massively parallel architectures

By

Swapan Mallick

June 1996

A thesis submitted for the Degree of
Doctor of Philosophy



Department of Engineering

University of Warwick

SUMMARY

To take advantage of the inherent flexibility of the finite element method in solving for flows within complex geometries, it is necessary to produce efficient implementations of the method. Segregation of the solution scheme and the use of parallel computers are two ways of doing this.

Here, the optimisation of a sequential segregated finite element algorithm is discussed, together with the various strategies by which this is done. Furthermore, the implications of parallelising the code onto a massively parallel computer, the MasPar, are explored.

This machine is of Single Instruction Multiple Data type and so modifications to the computer code have been necessary. A general methodology for the implementation of finite element programs is presented based on projecting the levels of data within the algorithm into a form which is ideal for parallelisation. Application of this methodology, in a high level language, has resulted in a code which runs at just under 30MFlops (in double precision). The computations are performed with minimal inter-processor communication and this represents an efficiency of 20% of the theoretical peak speed. Even though only high level language constructs have been used, this efficiency is comparable with other work using low level constructs on machines of this architecture. In particular, the use of data parallel arrays and the utilisation of the non-unique machine specific features of the computer architecture have produced an efficient, fast program.

Contents

Acknowledgements	vi
Declaration	vii
List of Figures	viii
List of Tables	xi
1 INTRODUCTION	1
1.1 Background	1
1.2 Motivation	5
1.3 Objectives	8
1.4 Presentation of Material	9
2 SOLVING THE NAVIER-STOKES EQUATIONS	10
2.1 Equations of Motion	10
2.2 Discretisation Methods	12
2.2.1 Finite difference method	12
2.2.2 Finite element method	14
2.2.3 Finite volume method	16
2.2.4 Spectral methods	17
2.3 Comparison of the Discretisation Methods	18
2.4 Considerations for Solving the Navier-Stokes Equations	20
2.4.1 Resolving non-linearity	20
2.4.2 Time dependence	21
2.4.3 Discretising the convection term	22
2.4.4 Alternative treatments of the convection term	22
2.4.5 Pressure problems	24
2.4.6 Overcoming the checkerboard problem	25
2.5 Finite Element solutions of Navier-Stokes equations	26
2.5.1 Primitive variable formulation	26

2.5.2	Penalty methods	27
2.5.3	Streamfunction and vorticity methods	27
2.5.4	Babuska-Brezzi conditions	29
2.6	SIMPLE: A Segregated Approach	29
2.6.1	The SIMPLE algorithm	31
2.6.2	Variants of SIMPLE	32
2.7	Segregated Finite Element Algorithms	35
2.8	Description of the Scheme	37
2.9	Overview of Algorithm	43
2.10	Input Data and Parameter Importance	45
2.10.1	Primary data	45
2.10.2	Secondary data	47
2.10.3	Tertiary data	47
2.11	Simple Test Cases and Code Validation	51
2.11.1	Validating the steady-state and time dependent terms	51
2.11.2	Establishing the directional integrity of solutions	52
2.12	Solution Strategies	53
3	ACHIEVING FAST SOLUTIONS	60
3.1	Case Studies	60
3.2	Profile of the Solver	61
3.3	Reducing Computational Overheads	61
3.3.1	Arithmetical rearrangement	62
3.3.2	Common sub-expression evaluation	64
3.4	Mass Lumping	65
3.5	Iterative Solvers	66
3.6	Preconditioners	70
3.7	Application of Iterative Solvers	71
3.8	Properties of Matrices	71
3.9	Storage Schemes and Memory Access	73

3.10	Discussion	74
4	PARALLELISATION CONSIDERATIONS	88
4.1	Parallel Architectures	88
4.2	Performance Measurements	90
4.3	Choice of Architecture	91
4.4	Manchester's MasPar Machine	92
4.5	Parallel Approaches to Finite Element Algorithms	93
4.5.1	Load balancing problems	94
4.5.2	Data storage	95
4.5.3	Solution of linear equations	96
4.5.4	Imposition of boundary conditions	98
4.6	Discussion	98
5	PARALLELISATION STRATEGY	102
5.1	Overview of SIMD problems	102
5.1.1	Scalability	102
5.1.2	Storage	103
5.1.3	Communication complexity	103
5.1.4	Computational complexity	105
5.2	Previous SIMD strategies	106
5.3	Measuring Communication Costs	107
5.4	Analysis of finite element algorithm	108
5.5	The Use of Data Levels	109
5.5.1	Definition	109
5.5.2	The data level concept applied to an algorithm	110
5.6	Parallelisation strategy	111
5.7	Application of parallelisation strategy	112
5.7.1	FE data levels	112
5.7.2	Communication costs of original algorithm	112
5.7.3	Selection of elementary object	113

5.7.4	Case (i): Levels N and S moving to level S ($N.S \rightarrow S$):	113
5.7.5	Case (ii): Levels E and N moving to level E ($E.N \rightarrow E$):	114
5.7.6	Case (iii): Levels G and N (with IPC') moving to level N ($G.N' \rightarrow$ N):	114
5.7.7	Case (iv): Boundary conditions: Level B moving to level N ($B \rightarrow$ N):	116
5.8	Final data level movements and communication costs	116
5.9	Results of Implementation	118
5.10	Discussion	119
6	APPLICATION OF PARALLEL STRATEGY	140
6.1	Introduction	140
6.2	Limitations of Incompressible Codes	140
6.3	Upwind Petrov Galerkin Method	141
6.4	Implementation Details	142
6.5	Problems of Parallelising Upwinding	144
6.6	Application of Parallel Strategy	145
6.7	Results and Discussion	145
7	VERIFICATION OF THE PARALLEL CODE	148
7.1	Introduction	148
7.2	Comparison of sequential and parallel code	148
7.3	Backstep flow	150
7.3.1	Flow domain and mesh considerations	151
7.3.2	Solution parameters	151
7.3.3	Results	152
7.4	Driven cavity flow	153
7.5	Cylinder flow	154
7.6	Discussion	156
8	MODERATE REYNOLDS NUMBER FLOWS	172

8.1	Introduction	172
8.2	Driven cavity flow	172
8.3	Cylinder flow	173
8.4	Calculations with upwinding	174
8.5	Discussion	176
9	CONCLUSIONS	190
	REFERENCES	193
A	Iterative linear equation solvers	201
A.1	Line relaxation methods	201
A.2	The conjugate gradient (CG) method and its derivatives	202
A.3	Minimum residual methods	205
B	Manchester's MasPar machine	206
B.1	Specifications	206
B.2	Data transfer and inter-processor communication	207
B.3	Programming Tools	207
B.4	Array Mapping	208
C	Mapping Variables on the MasPar	210

Acknowledgements

I would like to thank EPSRC (formally SERC) for funding this doctoral programme. Further, this work would not have been possible without the assistance of Dr. Jitesh Gajjar at the University of Manchester and Nigel Jagger of MasPar Co.

Special thanks must go to my supervisor, Dr. Chris Shaw, who has guided me throughout this work. His steadfastness and support have been sorely tried and tested.

During the course of this work, the Fluid Dynamics research group at Warwick University has created a highly stimulating environment. Thank you.

I dedicate this thesis to my father and mother. Only their belief in the value of education and their unceasing support made this possible.

Declaration

Some sections of Chapters 4 and 5 have been published in the following:

S. Mallick and C.T. Shaw,

Computationally Efficient Segregated Finite Element Algorithms

Proceedings of VIIIth Int. Cont. on Numerical Methods in Laminar and Turbulent Flow,

edited by C. Taylor, volume 8, part 2, pages 1517–1528.

Published by Pineridge Press

S. Mallick and C.T. Shaw,

Segregated Finite Element Algorithms on massively parallel machines

Proceedings of VIIIth Int. Conf. on Finite Elements in Fluids,

edited by K. Morgan, E. Onate, J. Periaux, J. Peraire, and O.C. Zienkiewicz, part 2, pages 1231–1240.

Published by Pineridge Press.

List of Figures

1	Central differencing of a first derivative of a continuous function.	56
2	A simple control volume with two cells.	56
3	Velocity field for simple parabolic flow.	57
4	Effect of rotation of a complete flow problem on the values of velocity at one point in the flow.	58
5	Effect of rotation of a complete flow problem on the values of pressure at one point in the flow.	59
6	Domination of solution time by linear equation solvers as problem size increases.	78
7	Effect of some computational reductions.	79
8	Comparison of execution times of solvers running with and without mass lumping.	80
9	An example of the sparsity pattern for 2D driven cavity problem.	81
10	Diagonal dominance ratings for velocity generated matrix.	82
11	Diagonal dominance ratings for a pressure generated matrix.	83
12	Ratio of eigenvalues for velocity generated matrix.	84
13	Ratio of eigenvalues for pressure generated matrix.	85
14	Diagonal dominance of velocity generated matrices.	86
15	Diagonal dominance of pressure generated matrices.	87
16	General layout of a MasPar.	101
17	Structure of the main program of the sequential implementation.	127
18	Typical structure of a sub-calculation of the sequential implementation. .	128
19	Strategy for parallelisation.	129
20	Structure of the main program of the parallel implementation.	130
21	Typical structure of a sub-calculation of the parallel implementation. . .	131
22	Part 1 of parallel subroutine code.	132
23	Part 2 of parallel subroutine code.	133
24	Part 3 of parallel subroutine code.	134

25	Part 4 of parallel subroutine code.	135
26	Part 5 of parallel subroutine code.	136
27	Part 6 of parallel subroutine code.	137
28	Solution times for various sized problems.	138
29	Solution times for iterative solvers on various sized matrices.	139
30	Calculation of the centroid of a quadrilateral element.	147
31	Calculation of the centroid of a hexahedral element.	147
32	Comparison of residuals of the solution of the u-velocity system of linear equations from the sequential and parallel codes.	157
33	Comparison of residuals of the solution of the pressure-generated system of linear equations from the sequential and parallel codes.	158
34	Backward-facing step geometry with boundary conditions.	159
35	Backward-facing step mesh layout.	159
36	Backward-facing step mesh.	160
37	Pressure increments for backstep flow.	161
38	U-monitor values for backstep flow.	162
39	Pressure monitor values for backstep flow.	163
40	Velocity field at the inlet of a backstep flow.	164
41	Pressure field at the inlet of a backstep flow.	164
42	Close up view of velocity field near step in backstep flow.	165
43	Velocity field at outlet for backstep flow.	166
44	Domain and boundary conditions for 2D driven cavity problem.	167
45	Driven cavity velocity field at $Re=10$	168
46	Cylinder flow mesh.	169
47	Cylinder flow mesh geometry.	169
48	Pressure contours for cylinder flow at $Re=5$	170
49	Velocity field for cylinder flow at $Re=5$	171
50	Velocity field for driven cavity flow at $Re=100$	180
51	Centreline u-velocities for driven cavity flow at $Re=100$ compared with results from Burggraf [85].	181

52	Cylinder flow mesh geometry.	182
53	Fine cylinder mesh layout.	183
54	Cylinder flow mesh biasing.	183
55	Coarse cylinder mesh.	184
56	Fine cylinder mesh.	184
57	Wiggles in solution when the convection term is discretised with central differencing.	185
58	Pressure field for cylinder flow at $Re=25$	186
59	Velocity field for cylinder flow at $Re=25$	187
60	Wake length behind cylinder flow at varying Reynolds numbers.	188
61	Divergence of monitor values for cylinder flow at $Re=50$	189

List of Tables

1	Types of data required by the implementation of Shaw [19,20] to solve a flow problem.	55
2	3D Driven cavity mesh statistics. These meshes are used to determine timing characteristics of a solution procedure over problems of varying degrees of freedom, representing the Case 1 set of problems for Chapter 3.	75
3	Pipe flow mesh statistics. These meshes represent the Case 2 set of problems for Chapter 3 and are used to compare accuracy with different solution methods.	75
4	Operation timings on various machines.	76
5	Monitor values for pipe flow experiments with no mass lumping.	77
6	Monitor values for pipe flow experiments with mass lumping.	77
7	Elapsed times (in milliseconds) for the calculation of a single matrix-vector product.	122
8	Processing speeds for the different sections of the shape function calculations.	122
9	Data level movements and inter-processor communication values of the original algorithm.	123
10	Elapsed times (in milliseconds) and communication costs for a single matrix-vector product calculation using the scatter-gather approach of Fischer and Patera [61].	124
11	Data level movements and communication costs for final algorithm. . . .	125
12	MFlop rates and execution times (as a percentage of total execution time) for each part of the final algorithm.	126
13	Data level movements for upwinding.	146
14	Data level movements for final upwinding.	146
15	Simple comparison of FORTRAN-77 and MPFORTRAN	208

1 INTRODUCTION

1.1 Background

Mathematical modelling of phenomena has allowed greater insight into the physical world. Contributions to the physics of a situation which can be considered negligible may be ignored, thereby simplifying any analysis. Whilst this is an essential practice, care must be taken to ensure that systems are not over-simplified, and to ensure that models which behave reasonably in certain situations are not used to analyse problems in which the underlying assumptions are untrue. The goal of a model is to accurately predict both quantitatively and qualitatively the behaviour of the system that it is supposed to be modelling.

The dynamics of fluids can be extremely complex, encompassing non-linear and chaotic states. Thus initial models were defined for specific problems. Scientists sought for many years for a general mathematical model which described fluid behaviour. Historical reviews from Acheson [1], Tritton [2] and Roberson and Crowe [3] have been compiled to give an overview of how models developed.

One of the first non-specific analyses of fluid behaviour came from Newton [4] in 1687. He defined the *resistance* of a fluid to be proportional to the velocity with which *parts of the fluid* are separated from one another. He was referring to the stress at any point within the fluid but did not have partial differential calculus with which to analyse this, and the concepts of fluid *parts* was not well defined. Nevertheless, a fluid in which the stress tensor and the rate-of-strain tensor are linearly related is termed Newtonian. Non-Newtonian behaviour may arise in fluids where the molecular structures are gathered and organised in some sense, for example in polymers, suspensions and emulsions.

In 1743, Bernoulli [5] defined the concept of internal pressure. Then in 1752, Euler extended Newton's principle of the conservation of linear momentum to any infinitesimal body within an elastic or fluid medium. This he combined with the concept of internal

pressure to obtain the (Euler) equations of motion for an inviscid fluid. At the same time he formulated the calculus of partial derivatives. In 1822 Cauchy introduced the concept of the *stress tensor* and combined it with Euler's laws of mechanics to construct a general theoretical framework for the motion of any continuous medium. Thus to study a Newtonian viscous fluid it only became necessary to add an appropriate constitutive relation describing its physical properties. In 1845, Stokes extended Newton's original laws of motion to obtain the appropriate constitutive relation, so deriving the Navier-Stokes equations. Note that the double-barrelled naming is due to the earlier work of Navier which led to the correct equations, unfortunately from invalid assumptions about the molecular basis of viscous effects (see Acheson [1]).

The Navier-Stokes equations are a mathematical statement relating principle variables of interest, that is the velocity and pressure. These are known as *primitive variables* (or *primitives*). Depending on the nature of the situation under examination, other primitive variables may be included via the addition of extra terms and equations to include, for example, the effects of heat transfer where variables such as temperature and enthalpy are used.

For many simple flows (requiring simple domains, initial and boundary conditions such as the steady incompressible flow between two infinite parallel plates) it is straightforward to obtain analytic solutions, and these are well documented (for example Acheson [1] and Tritton [2]). Analytical solutions are desirable because they give solutions over a continuum within the domain of interest within which they are valid. However, for flows in general, it is usually impossible or impractical to obtain an analytic solution, and so instead some form of numerical solution must be obtained.

A numerical solution differs from an analytical solution in that values of primitive variables may not be explicitly determined over the entire domain. Instead, the solution may be determined at discrete points within the domain under consideration. These points may be called *nodes*, or *vertices*. These define the computational *mesh* or *grid* which may or may not be static in the domain. The governing equations may be solved in a

piecewise manner across *elements* or *cells*, which cover the domain completely, without overlapping each other. These *element equations* may be combined at some point in the solution procedure to link the various piecewise solutions together in a meaningful manner, thereby coupling the piecewise solutions across the domain (see Hirsch [6]).

An alternative approach is to assume that the equation has a solution of some predetermined form, which usually consists of a sequence of expressions with unknown constants. Spectral methods use this approach (see Canuto *et al* [7]). The task is then to evaluate these coefficients to give a solution over the domain.

The solution scheme may be explicit or implicit. In explicit schemes, unknown primitives are expressed in terms of known values. Such schemes have been popular because they are simple to program and are widely used for calculations of inviscid flows. However, explicit methods can be unsuccessful for viscous flows because of the Courant-Friedrichs-Lewy (CFL) stability condition that may require a prohibitively small time step to maintain stability of the numerical scheme (see Hirsch [6]). Conversely, implicit schemes express unknown primitives either entirely by other unknown variables or a mixture of known and unknown variables. Though it would not be possible to evaluate a primitive at one particular node with just one implicit equation, the combined set results in a closed algebraic system of equations which may be solved simultaneously. Such schemes have the disadvantage of requiring storage for the algebraic system but do not suffer from the restrictive time step requirements that apply to explicit schemes.

The earliest and simplest method of solving differential equations numerically is the finite difference (FD) method. This is first thought to have been used by Euler in 1768. However the time required to carry out solutions was prohibitive until 1950 when the electronic computer made high speed calculations possible. Los Alamos Scientific Laboratory, through the efforts of Neumann gave considerable impetus to computational fluid dynamics (CFD) as a science in its own right; in 1965 Harlow and Fromm [8] published an article in Scientific American that created widespread awareness and interest in the utility of CFD. They investigated air flow past a rectangular rod, using 1176 cells on a

regular mesh using both an IBM 7090 and a purpose-built machine known as STRETCH. Even in this very coarse model, the qualitative characteristics of vortex shedding could be observed. Soon after, in 1972, the FD algorithms were adapted to the more general finite volume (FV) method of Patankar and Spalding [9]. This is probably the most popular method in CFD.

The finite element (FE) method is another procedure for solving differential equations. Its chronology does not extend from FD or FV methods, but instead from the concept of variational forms. The idea of using variational principles and approximations by smooth piecewise functions, was used by Leibnitz in 1696 (in one dimension with piecewise linear functions). In two dimensions, triangulation and piecewise linear functions were used by Schellbach [10]. Though modern FE techniques began with the paper by Turner *et al* [11], the expression *finite elements* was introduced by Clough [12] in 1960.

The FE method has been extensively used in structural analysis problems (for example Hoff [13]). Its ability to deal with complex geometrical shapes was ideal for problems of this nature. This had appeal to the CFD community and was adopted for this, and other fields. Problems of complex geometries exhibit several characteristics: they often do not conform to simple co-ordinate systems, they may exhibit steep variations in wall boundary layers which are difficult to resolve numerically, and they may contain embedded thin shear layers, which are also difficult to resolve.

Current computer hardware and algorithmic improvements allow two dimensional steady-state Euler solutions to be obtained in seconds, a task which would have taken hours in the 1970's. Today, Euler codes are well established and may be applied to complex configurations, such as the flow over a complete aircraft and the flow through turbomachinery (see Hirsch [14]).

Industrial interest in CFD stemmed from the possibility that the costly process of building and rebuilding physical models could be reduced. A commonly quoted example is the dramatic Tacoma Narrows bridge which was shaken to pieces by wind-induced vibrations

soon after it was built (see Roberson and Crowe [3]). In the process of rebuilding it, many elaborate models were built and tested repeatedly. The time and effort taken could have been reduced had CFD techniques then been available and practical. Now many commercial packages are available which are capable of solving a wide range of flow problems. Turbulence modelling, for example, is a common feature of many such packages.

However, it would be naive to assume that CFD can altogether negate the need for experimental fluid dynamics. Whilst there are advantages in obtaining numerical values of primitive variables without affecting the flow field with some measuring apparatus, approximation and discretisation errors are unavoidable when implementing a numerical algorithm. Nevertheless, CFD provides a useful tool in validating theories and gaining insight into the processes that occur in fluids. Moreover, CFD has a permanent place as a design and predictive tool in both qualitative and quantitative ways.

Fluid dynamics is not merely about solving the NS equations for a particular problem. In real problems it is much more usual to have a large parameter space that precludes purely numerical solution, and so analytical, heuristic and experimental data, as well as intuition, must be used for satisfactory solutions to be obtained.

1.2 Motivation

In less than 50 years, CFD has advanced from infancy to industrial use and yet there are still academic and industrial applications that remain unfeasible due to the machine limitations. For example, flows involving transition have length scales that are tiny compared to those of laminar flow, requiring increased numerical resolution. Solution of such flows require increased computational costs both in terms of storage and time to solution. For this reason, few of the commercial packages available offer the analysis of flows undergoing transition, a subject currently under intense academic investigation. Such problems are of interest for a variety of reasons; drag forces and heat transfer rates are

so much higher in turbulent boundary layers that the ability to control transition would open up significant possibilities to a design engineer. Delaying transition is important, for example, for a space vehicle re-entering the earth's atmosphere to reduce the heat transfer onto the surface of the vehicle. Alternatively, one may want to deliberately trigger transition as early as possible in order to delay boundary layer separation, thus minimising the size of the wake produced and thereby reducing drag. For this reason, golf balls have dimples. A numerical solution of such problems is not usually evaluated because of the length of computer time required. However, this could change if the efficiency of algorithms is improved and/or processing power increases dramatically.

One may improve implementations of existing algorithms in various ways: by reducing computational overheads, modifying existing algorithmic techniques which are less computationally demanding or by developing techniques specific to solving particular problems. Computational overheads arise from a variety of sources within an implementation. For example, arithmetic statements may be processed in different ways without modifying solutions (discounting round-off errors). Moreover, the data structures within an implementation affect the way data is accessed. Also, replacement of techniques within sub-problems of an algorithm with faster alternatives may lead to quick solutions. On the other hand, techniques specific to particular problems have been a common method of reducing processing time. For example, in the solution of turbulent flow problems, simplifying the governing equations by including Reynolds stress terms has been a popular approximation to the full Navier-Stokes equations allowing significant reduction of computing time.

This has led to the introduction of various turbulence models. A review of these may be found in Bradshaw *et al* [15]. However, such models require additional data which may only be determined experimentally as the resulting equations are not algebraically closed. This is referred to as the closure problem. This approach means that turbulence coefficients for one type of problem are not usually valid for other problems.

The Navier-Stokes equations together with appropriate initial and boundary conditions

mathematically encompass all the processes involved in isothermal transition, yet obtaining numerical solutions has proved difficult. Thus it is desirable to have a solution procedure based on the full Navier-Stokes equations.

Discretisation methods for the solution of the full Navier-Stokes equations have been commonly based on FD or FV schemes. However, the solution procedure should be able to cope with a wide range of geometries, without the restriction of requiring topologically cuboidal meshes as has been the case with classic FD and FV schemes. Thus the discretisation procedure selected for this work is a FE scheme, specifically to solve the full Navier-Stokes equations.

However, whilst all three of these grid dependent schemes have been popularly used, the FE method is relatively slow. This is due to the fact that whilst the FD and FV methods use the mesh topology from which to calculate intermediate equations, the FE method uses only the geometry of the element in question, and only later considers that they are all joined together; moreover, FD and FV schemes often use lower-order integration. This is also why an FE scheme has greater generality, and is more amenable to complex geometries. Moreover, it is believed that the most general and reliable incompressible viscous flow simulators are based on FE methodologies, and are well suited to industrial applications (see Glowinski and Pironneau [16]). Given the restriction of time step on explicit methodologies, an implicit scheme is desirable.

A problem with this approach is the processing power required. Even though computational performances have increased dramatically, sequential computers are approaching the limit of their processing speed, and are not able to provide necessary processing power (see Quinn [17]). Thus processors have been linked together in a parallel fashion to provide a combined overall increase in power. Different architectural possibilities for a parallel machine exist, first categorised by Flynn [18]. Ideally, the processing speeds of machines are *scaleable*, that is proportional to the number of processors. In practice, it is not possible to attain theoretical performances because it may not be possible to ensure that all processors are working constructively all the time (a *load balancing* prob-

lem), and because communication between processors will usually be necessary, further reducing parallel efficiency. Minimisation of these costs is a major problem. Whilst it is not too difficult to implement an algorithm, on certain parallel architectures, to obtain an efficient implementation is much more difficult.

1.3 Objectives

Considering the algorithmic and hardware limitations that current CFD techniques have, there is considerable scope for improvement. Many discretisation methods have been developed in the past, so one is able to select the most appropriate for specific classes of problems. We have selected the implicit segregated FE algorithm of Shaw [19], [20] which solves the full incompressible Navier-Stokes equations for Newtonian fluids. This algorithm automatically fulfils the criteria of low storage requirements whilst retaining an implicit methodology, allowing complex geometries to be modelled by virtue of its FE formulation.

Given the widely differing types of parallel machines available, it is not possible to construct a generic methodology. Instead, we have selected one class of parallel architecture, and seek to determine implementation strategies. The aims of this work are therefore:

1. To produce strategies for efficient solutions to FE algorithms on sequential architectures.
2. To produce implementation strategies for one class of parallel machines. This strategy should be general enough to allow guidance for the implementation of FE algorithms on similar architectures.
3. To demonstrate the usefulness of the final implementation on a practical problem.

1.4 Presentation of Material

In Chapter 2, the Navier-Stokes equations are discussed in some depth, together with the computational challenges that their numerical solution presents. Common discretisation methods are discussed and compared; in particular, FE methodologies are considered. The solution scheme that forms the starting point for this work is then detailed, and its validity demonstrated. We then consider, in Chapter 3, how such an implementation may be optimised on sequential architectures. General strategies are presented which are not limited solely to FE codes in principal.

In Chapter 4, we discuss parallel architectures, and the considerations necessary for implementing codes. A parallelisation strategy is presented in the subsequent chapter, which is general in approach yet allows efficient implementations. In Chapter 6, a modification of the formulation is implemented as an extension of the original algorithm. This allows the solution of moderate Reynolds numbers to be generated.

In Chapter 7, a detailed validation of the parallel implementation is presented, with some solutions to low Reynolds number flows. Comparisons are made between the solution times using both the sequential and parallel implementations. In the following chapter, the parallel code is exclusively used to generate solutions at moderate Reynolds numbers. Upwinding is used to discretise the convective terms appropriately, and the limitations of the implementation are discussed. Chapter 9 is the conclusions chapter which sums up the work presented.

2 SOLVING THE NAVIER-STOKES EQUATIONS

2.1 Equations of Motion

The derivations of the equations of motion are well reported (for example, Tritton [2], or Hirsch [6]), and so we shall simply state them in the incompressible form, in the absence of body forces:

$$\rho \frac{\partial u}{\partial t} + \rho u \cdot \nabla u = -\nabla p + \mu \nabla^2 u \quad \text{in } \Omega \quad (1)$$

$$\nabla \cdot u = 0 \quad \text{in } \Omega \quad (2)$$

where Ω denotes the domain of the flow, and is a subset of \mathbf{R}^d , where $d = 1, 2, 3$. This is contained by the boundary Γ , a subset of \mathbf{R}^{d-1} . The values u, p and μ are the velocity vector, the scalar pressure field and the viscosity respectively.

Equations (1) and (2) are known as the momentum and continuity equations respectively, and together constitute the incompressible Navier-Stokes equations. Each of the terms in these equations has a meaningful physical interpretation. Reading from left to right, on the left hand side of the momentum equations, is the acceleration (or time dependent) term and the convection (or inertia, or advection) term. Then the right hand side consists of the pressure term and the viscous (or diffusion) terms. The continuity equation states that mass is conserved within any finite volume.

The nature of the Navier-Stokes equations changes depending on which forces are dominant within the flow. One useful measure of the dominant forces is the Reynolds number defined by the equation:

$$\text{Re} = \frac{\rho}{\mu} U L \quad (3)$$

where U and L are representative velocity and length scales.

At low Reynolds numbers, viscous forces dominate the flow and the equations reduce to the Stokes equations:

$$\rho \frac{\partial u}{\partial t} - \mu \nabla^2 u = -\nabla p \quad (4)$$

Equation (4) is elliptic for steady flows for a fixed pressure gradient, and can be physically interpreted as a diffusion problem. Note that the Laplacian and Poisson equations are the simplest representations of this phenomena. However, when the flow is unsteady, the equation becomes parabolic due to the time derivative. Parabolic equations are mathematical models for systems in which the diffusion process occurs but is damped in time. On the other hand, when the Reynolds number is high and the inertial forces dominate, the NS equations reduce to the inviscid Euler equations (outside regions of high viscous shear):

$$\rho \frac{\partial u}{\partial t} + u \cdot \nabla u = -\nabla p \quad (5)$$

Equation (5) is hyperbolic and can be physically interpreted as a wave propagation problem. Moreover, the continuity equation (2), which states that mass cannot be created or destroyed (in the absence of source or sinks which may be present, for example, due to multiphase situations in which chemical reactions occur) is also hyperbolic in nature. Thus at low Reynolds numbers, the flow is elliptic-hyperbolic for steady problems and

parabolic-hyperbolic for unsteady problems, but at high Reynolds numbers, the flow is hyperbolic.

The classification of PDEs into hyperbolic, parabolic or elliptic provides information on the nature of the characteristics of the equations; that is, on their curves of information propagation. Computationally however, this classification is not as important as the classifications into initial value (or marching, or Cauchy) problems (which are either parabolic or hyperbolic), and boundary value problems (which are elliptic) (see Press *et al* [21]). We will be examining both steady-state and transient problems; that is, both boundary value and initial value problems.

In general, the overriding consideration for initial value problems is stability of the algorithm, whereas for boundary value problems, efficiency is the overriding consideration as stability is easier to guarantee (see Press *et al* [21]). However, for our FE schemes (to be discussed in Section 2.7), the acceleration terms actually improve the stability of the algorithm (see Shaw [19]). This will be discussed in Section 2.4.2.

2.2 Discretisation Methods

As discussed in Chapter 1, four discretisation methods are commonly in use. These are the FD, FV, FE and spectral methods, and are considered in the following sections.

2.2.1 Finite difference method

The FD method is based on approximating partial derivatives with appropriate truncated expressions based on the Taylor series in order to allow a difference equation to be formed. It is the oldest of the discretisation methods. From a given initial condition, it should then be possible to evaluate all the primitive variables in the problem, at the preselected grid points. The main advantage of the FD method is its high speed. This results from the fact that the derivatives are calculated from the regular geometry of a structured

mesh, and coded into the program. This means that in a sense, some of the calculation work is done before running a problem, but the method has the limitation that if a mesh is changed then the code has to be reformulated. Some flows occur in geometries which are so complex that not even a grossly distorted topologically regular mesh can adequately model them. For problems of this kind, an alternative scheme, for example the FE method must be used.

Consider a point x , and two points a small displacement h either side of it, at $x + h$ and $x - h$, as shown in Figure 1. For a general function $U(x)$, the Taylor series expansions at the two points are:

$$U(x + h) = U(x) + \frac{h}{1!} \frac{dU}{dx} + \frac{h^2}{2!} \frac{d^2U}{dx^2} + \frac{h^3}{3!} \frac{d^3U}{dx^3} + \dots \quad (6)$$

$$U(x - h) = U(x) - \frac{h}{1!} \frac{dU}{dx} + \frac{h^2}{2!} \frac{d^2U}{dx^2} - \frac{h^3}{3!} \frac{d^3U}{dx^3} + \dots \quad (7)$$

Simple algebraic manipulation of these equations leads to the following expressions for the derivatives:

$$\frac{dU}{dx} = \frac{1}{2h} [U(x + h) - U(x - h)] + O(h^2) \quad (8)$$

and

$$\frac{d^2U}{dx^2} = \frac{1}{h^2} [U(x + h) - 2U(x) + U(x - h)] + O(h^2) \quad (9)$$

where $O(h^n)$ refers to terms of order n or higher.

Thus, for small h , these terms should be negligible, and equations which are n^{th} order accurate are ones which neglect only the terms of order n or higher.

Note how the denominator in the equation for the first derivative, is $2h$. Intuitively, this formula is defining the gradient of U at a point x , by looking at the values at the two points either side of it and finding the straight line gradient between the two. Hence, this is known as the centred differences formula for the first derivative.

However, this equation for the first derivative is by no means unique, and others may be defined:

$$\frac{dU}{dx} = \frac{1}{h} [U(x+h) - U(x)] + O(h) \quad (10)$$

(called the forward difference formulation),

$$\frac{dU}{dx} = \frac{1}{h} [U(x) - U(x-h)] + O(h) \quad (11)$$

(called the backward difference formulation).

Using equations such as these, and appropriate initial conditions, the value of a variable at each point in the mesh can be expressed in terms of the values at surrounding points. This leads to either implicit or explicit general equations for the entire domain, which can easily be solved.

2.2.2 Finite element method

The finite element (FE) method solves variational forms of a differential equation in a piecewise manner over each element in the domain. The *order* of each element is defined by the degree of the polynomial, used by that element to approximate the primitive

variables resulting in some equation. In implicit methods, these *element equations* may then be assembled together so that a *global stiffness matrix* and *force vector* is obtained. Boundary conditions are then imposed on the matrix, and a solution for the primitives may be obtained.

Most fluid dynamics problems requiring the solution of differential equations have normally been solved with FD methods, using topologically cuboid meshes. However, as CFD increases in its applicability, the desire for a FE solution scheme suitable for complex geometrical problems has been growing. Thus several schemes have been designed and we will discuss some of these later in Section 2.5. The basic methodology of the FE method is given below and many texts are available which give a more detailed mathematical description, for example Zienkiewicz and Taylor [22] and Reddy [23].

The methodology for an analysis with the FE method is as follows:

1. Discretisation of the domain into a set of FEs (or geometric shapes), labelling each node (in the element), and labelling each element. A matrix containing information about the nodes which define each element, called the *connectivity* matrix, is necessary.
2. Derivation of element equations for all typical elements in the mesh. This is done by constructing the variational formulation of the differential equation to be solved over each typical element, deriving or finding the element interpolation functions ψ_i , with the assumption that each dependent variable u is of the form

$$u = \sum_{i=1}^n u_i \psi_i \quad (12)$$

where n is the number of equations in the mesh, and u_i is the value of each dependent variable at node i .

3. Assembly of the element equations to obtain a global matrix equation for the whole problem

4. Imposition of the boundary conditions
5. Solution of the global equations
6. Postprocessing of the results

The main reason that the FE method has not been the preferred method of dealing with CFD problems is its comparatively slow speed. Traditional methods have used coupled solutions and direct solvers to solve linear systems of equations, which are partly responsible for the length of time required to generate solutions. However, with increasing computer power and the implementation of iterative methods to solve the simultaneous sets of equations that arise, the FE method is becoming increasingly attractive.

2.2.3 Finite volume method

The FV method is an alternative discretisation procedure (see Patankar [24], Gosman *et al* [25] or Versteeg and Malalasekera [26]). In this method, the domain is discretised into a set of nodes. However, nodes are not placed directly on the boundary. Then, each node is considered to be surrounded by a control volume (or cell). The boundaries of the control volumes are positioned mid-way between adjacent nodes. It is usual to arrange this discretisation so that the physical boundaries of the domain to be analysed coincide with the control volume boundaries.

The key step of the FV method is to integrate the governing equations over a control volume. This gives a discretised equation at each nodal point of the control volumes. For control volumes which have a face coincident with the domain boundaries, the discretised nodal equations are modified to incorporate boundary conditions. This results in a closed set of algebraic equations which may easily be solved.

2.2.4 Spectral methods

The spectral method is another discretisation procedure, in which the variation of each dependent variable is represented by an infinite series (for example, Fourier series or Chebyshev polynomials) truncated to N terms (see Canuto *et al* [7]). The series is assumed to apply over the entire domain, and substitution of the series into the governing differential equation produces a global equation involving all the coefficients of the series. Then, expressions for the N coefficients are obtained by imposing one of several constraints (for example, collocation constraints - which satisfy the equations at discrete points in space). Specialised methods such as fast Fourier transforms often have to be used to determine the non-linear products arising from the convective terms.

This is a fairly complex method which does not always work, even though the method in principle is not restricted by geometry. This is because it is very sensitive to the type of boundary conditions applied. For example, Fourier series work only with periodic boundaries. However, when the method does work, a very high spatial accuracy can be attained (exponential with N), but at the cost of very high computing effort and high storage requirements. Also, physical constraints are not always very easily enforceable, if at all.

The spectral method is an excellent choice when a problem can be fitted to it because of its high accuracy and efficiency, but if not then it is very difficult to implement. Thus for problems in simple geometries, it is excellent, especially those with periodic boundary conditions. However, for complex geometries it is very difficult to implement, though some work has been done on domain decomposition methods. For this reason, the method is typically used in problems where high accuracy is essential in simple geometries, for example in problems involving transition.

Babuska and Suri [27] have proposed two approaches which combine the flexibility of the FE method and the high rates of convergence of spectral methods. This combination is known as the p - and $h - p$ version of FE method. If the mesh is fixed, and greater

accuracy of the solution is achieved only by increasing the degree of the elements then we have the p -version. If the element degree remains constant and the mesh made finer then we have the h -version. If the mesh is refined and the elements are increased in degree then we have the $h - p$ version.

2.3 Comparison of the Discretisation Methods

The four major discretisation techniques have some basic similarities as well as some differences, though the spectral method is the most unusual.

Whilst similarities and differences are outlined below, the spectral method is not considered unless explicitly stated. The following similarities are noted by Hirsch [6] and Shaw [28]:

1. *Space discretisation* is required for all methods, where a mesh or grid is used to replace the space continuum. Hence, a finite number of points are defined at which the numerical values of the variables must be determined.
2. *Equation discretisation* is required by which the physical equations are transformed into an algebraic system of equations. This is also true of the spectral method.
3. For all four methods, a set of initial conditions is required, from which the solution may begin. Moreover, a set of boundary conditions is required in order to determine the values of the variables at the boundaries. This can be problematic, especially if the domain terminates amid non-trivial behaviour, for example, in a vortex street.
4. Either explicit or implicit forms of the equations may be produced, and simultaneous equations will need to be solved if an implicit form is produced.

Differences between the discretisation methods are found to be:

1. Both the FD and FV methods produce numerical equations at a given node, based on the values at neighbouring points. The FE method and the spectral method however, produce independently piecewise equations on an element. Only when these are assembled together is global interaction considered.
2. Whilst Dirichlet boundary conditions may be programmed directly into both the FD and FV methods, Neumann conditions are much harder to enforce.
3. The FD and FV methods are highly developed in the fluid mechanics field, because they are simpler to implement.

The FD and FV methods are fast, and appropriate for analysing well-defined problems for which mesh requirements are known in advance. The reason for this is that equations in this method are written to include the geometrical information of the problem. Mesh refinement, or alteration of the geometry of the domain requires rewriting the expressions for the equations, and may be time consuming. Some commercial packages overcome this by having a user interface within which a problem is defined. Then, a self-contained program is compiled within the package which solves the defined problem.

The FE and spectral methods, in initially producing independent piecewise equations allow problem geometry and mesh refinement without requiring code recompilation. Both methods are regarded as computationally expensive, but the FE method is respected for its ability to compute solutions in complex geometries, whilst the spectral method has been a useful academic method because of its high accuracy.

2.4 Considerations for Solving the Navier-Stokes Equations

Each of the discretisation techniques discussed in Section 2.2 may be used to solve a wide range of equations. However, calculation of the solution for the Navier-Stokes equations requires several special considerations because of the non-linearity of the equations, the time dependency and the problems involved in discretising the convection term. These issues will be discussed in the remainder of this section. The significant problem of producing an algorithm which allows the pressure to be derived is problematic because there is no pressure term in the continuity equation. Since this feature of the incompressible Navier-Stokes equation is largely responsible for the approach taken, we postpone discussion of this to Section 2.5.

2.4.1 Resolving non-linearity

The non-linearity is brought into the equation by the convection term, and is responsible for much of the complexity of the solutions. Because it is not possible for computers to deal with non-linear equations directly, we have to find a way of linearising the equations, into the form

$$Ax = b \tag{13}$$

where A is a known matrix, x is a vector containing the required primitive variables (which are initially unknown), and b is a known vector of functions of the flow variables.

This linearisation may be performed by discretising the derivatives of the convection terms as normal and replacing the pre-multiplying velocity variable by the current solution for the velocity values. For example, using the FD method on a regular grid and, using the central difference approximation for the first derivative,

$$u \frac{\partial u}{\partial x} \approx u_{i,j}^* \left[\frac{u_{i+1,j} - u_{i-1,j}}{2\delta x} \right] \quad (14)$$

where $u_{i,j}^*$ is the current value of u at a node in position (i, j) , and where $u_{i+1,j}$ and $u_{i-1,j}$ are neighbouring values of u , separated by a distance $2\delta x$.

This allows a linearised simultaneous set of equations to be formed and, by using a suitable technique (for example Gaussian elimination, or an indirect iterative solver such as the Gauss-Seidel method), a solution can be found to update the values of the flow variables. This linearisation and solution of simultaneous equations is repeated until the values of the primitive variables have converged.

This method works well for low Reynolds numbers, but it can produce odd-even decoupling (or checkerboard solutions). This is discussed in Section 2.4.6. However, for problems with larger Reynolds numbers, the term $u \frac{\partial u}{\partial x}$ needs to be discretised in different ways. These discretisation schemes are discussed in Section 2.4.3.

2.4.2 Time dependence

The time dependent term is easily discretised as it is a simple first derivative. From an initial condition, the solution proceeds from one time step to the next, after the primitive variables have converged within each time step.

For steady-state problems, the time loop could be left out of the solution procedure. However, not including the time dependent terms in the discretised form of the momentum equations can lead to numerical instability and cause the solution to diverge. Since the non-linearity of the equations requires a loop anyway, there is little advantage to leaving the time dependent terms out, so for steady-state cases, it is usual to leave these terms in, and use a very large time step to mimic the steady-state problem, thereby increasing stability (see Shaw [19]).

2.4.3 Discretising the convection term

Using a central difference discretisation method for the convection can present problems of divergence for problems where the Reynolds number is not small. The problem can be explained by examining the Peclet number for a cell, defined by Zienkiewicz and Taylor [22]:

$$Pe = \frac{\rho}{\mu}UL \quad (15)$$

where ρ is the density, μ is the viscosity and U and L are representative element velocity and length scales.

For problems in which the maximum Peclet number is greater than 2, convergence problems may occur. This may be avoided by using a first order difference equation instead of the second order central difference formula to model the first derivative. However, as a less accurate formula is used, a solution is produced which suggests more viscosity than the actual model. Nevertheless, this is a common method for resolving the problem, even though other schemes exist. These are discussed below.

2.4.4 Alternative treatments of the convection term

Courant *et al* [29] suggested using a lower order accuracy for the discretisation of the convection term. The motivation for this approach is that in convection dominated problems, information is propagated in the direction of the velocity. A number of schemes are now available, as discussed below.

1. **Upwinding of the convection term** thus calculating the derivatives of the convection term from only the primitives at the point of interest, and its nearest neighbour in the upstream direction. This usually allows resulting solutions to converge

but they may be inaccurate (see Hirsch [14]).

2. **Exponential scheme** which produces an exact solution for any value of the Peclet number, for any number of nodes for steady one dimensional problems. However, it is not widely used because the scheme requires a lot of computer effort, and the scheme is not exact for two or three dimensional situations; hence the extra computing power cannot be justified in light of the other available schemes (see Patankar [24]).
3. **Hybrid scheme** in which the upwind scheme is used if the Peclet number is greater than two, and the central difference equation is used if the Peclet number is two or less. Though this scheme is more accurate than (1), it does not converge for some meshes (see Shaw [20]).
4. **Quadratic upwind differencing (QUD)** also known as (**QUICK**) which is simply a quadratic upwind scheme. Though this is more accurate than either (1) or (3), numerical problems can be manifest for problems with complex geometries (see Shaw [20]).
5. **Power-Law scheme** which is based on QUD, and is more accurate. (see Patankar [24]).
6. **Upwind Petrov Galerkin (UPG) methods** which are based on normal upwind methods except that the convection term is weighted in the direction of the streamline for each element (see Brooks and Hughes [30] or Zienkiewicz and Taylor [31]).

Of these methods, the UPG method seems superior. It combines the motivation of weighting the convection term in the direction of information propagation, with reasonably low computer effort. Since some form of upwinding is necessary to allow solutions to be obtained at moderate to high Reynolds numbers, we have selected this method for application. A detailed consideration of this method is postponed until Chapter 7.

2.4.5 Pressure problems

The pressure gradient is manifest as part of the source terms of the momentum equations. In order to discretise the x -direction momentum equation, we must find some way of discretising the $-\frac{\partial p}{\partial x}$ term. Consider the domain shown in Figure 2.

We can find the centred-differences of the pressure from the half-way points shown by the points w and e . Hence,

$$-\frac{\partial p}{\partial x} = \frac{p_w - p_e}{x} = \frac{1}{x} \left(\frac{p_W + p_P}{2} - \frac{p_P + p_E}{2} \right) = \frac{p_W - p_E}{2x} \quad (16)$$

This equation could be derived simply by considering the pressure to be centred-differenced across points W and E .

Thus, the pressure term is actually represented by alternate grid points, and not two adjacent ones. This has two major implications. Firstly, the pressure term is derived from a coarser grid than the velocity terms, and secondly, a checkerboard pressure field may result (in which the pressure appears to oscillate). For a one dimensional pressure field, the fact that the field alternates heavily in a highly non-uniform manner, matters not, for the effect on the momentum equations will be zero, because by equation (16), there is no pressure force anywhere. This is also true for two or three dimensional pressure fields, which alternate in this manner.

Should any of these fields naturally occur during iterations, they will remain and result in an inappropriate solution. It is important to realise that a different placing of the control volume would make little difference; checkerboarding would still be manifest.

This problem is also manifest when we try to discretise the continuity equation. For example, in the steady one dimensional incompressible case, the continuity equation is just:

$$\frac{du}{dx} = 0 \quad (17)$$

Integrating over the control volume in Figure 2, leads to the equation below, where u_E and u_W refer to the velocities at E and W respectively.

$$u_E - u_W = 0 \quad (18)$$

Again, the equation requires equality of the velocities at alternate points, but not at adjacent ones, so a velocity field which alternates from point to point would satisfy the discretised equation, though it clearly does not satisfy the original equation.

2.4.6 Overcoming the checkerboard problem

A simple remedy to the checkerboarding problem is to use a staggered grid. There is no reason why we should be forced to calculate the values of the primitive variables all at the same point, so we can use a different grid for each variable should we want, and if this is done then the checkerboarding can be completely negated (see Harlow and Welch [32]). By calculating the velocity components on the faces of the control volumes, alternating solutions will not be permitted because the first derivatives are calculated by values from adjacent points and not alternate ones. The major disadvantage from this procedure is that the indexing and geometric locations of the velocity components must be stored, but in general, the method provides a practical technique for overcoming checkerboarding.

Another way to avoid this problem is to use a different form of discretisation (for example upwinding).

2.5 Finite Element solutions of Navier-Stokes equations

One major problem in solving the incompressible form of the NS equations can be seen from looking at the distribution of primitive variables in the momentum and continuity equations: in three dimensions, there are four equations and four unknowns, but the pressure is not specified in the continuity equation. It is indirectly: if a given pressure field is substituted into the momentum equations, and the velocity solved, then the resulting velocities satisfy the continuity equation. However, this is not very helpful for producing a computational scheme.

One method of overcoming this, for non-transient flows, is to add an artificial compressibility term to the continuity equation. When steady-state is reached, this term becomes zero; this is known as the pseudo-compressibility method of Chorin [33]. Transient methods are considered in the following subsections.

2.5.1 Primitive variable formulation

In the primitive variable formulation (also known as the pressure-velocity model), the primitive variables are discretised in a manner such that a single global coefficient matrix and resultant equation system is obtained. After applying appropriate boundary conditions, the system is solved for all unknown primitives in a simultaneous manner. Thus all the unknown velocities and pressures, and any other transported scalar variables which may be present in the problem are simultaneously updated.

Most of the popular methods that use the primitive-variable formulation are based on the separation of velocity and pressure. In this type of formulation as discussed by Ramaswamy and Jue [34], the incompressibility constraint is indirectly satisfied by solving the Poisson equation for the pressure. This is known as the pressure projection method. This methodology has the disadvantage of producing large equation systems, requiring storage. Moreover, the addition of other variables into the algorithm increases the size

of the system.

2.5.2 Penalty methods

In penalty methods, the continuity equation is treated as a constraint on the velocity components. The method involves the use of Lagrange multipliers, which represent the negative of the pressure (see Reddy [23]). However, a penalty approach leads to a non-symmetric definite ill-conditioned matrix with large condition number (see Haroutunian *et al* [35]). This makes the systems of linear equations that are produced difficult to solve.

2.5.3 Streamfunction and vorticity methods

A vorticity transport equation may be derived from the momentum equations for an incompressible fluid:

$$\frac{D\zeta}{Dt} = \frac{\partial\zeta}{\partial t} + u.\nabla\zeta = \zeta.\nabla u + \nu\nabla^2\zeta \quad (19)$$

where ζ is the vorticity, and ν is the kinematic viscosity. Moreover, the term $u.\nabla\zeta$ is the rate of change of vorticity due to convection, whereas $\zeta.\nabla u$ represents the rate of deformation of the vortex lines (and exists only in three dimensional flow). The last term represents the diffusion of vorticity.

For two-dimensional problems, this reduces to:

$$\frac{D\zeta}{Dt} = \nu\nabla^2\zeta \quad (20)$$

and the continuity equation becomes:

$$\nabla^2 \Psi = -\zeta \quad (21)$$

where Ψ is the streamfunction.

The dependent variables are streamfunction and vorticity; the velocities may be calculated from the streamfunction using:

$$u = \frac{\partial \Psi}{\partial y} \quad (22)$$

$$v = -\frac{\partial \Psi}{\partial x} \quad (23)$$

This approach has the advantage of not including the pressure term, and results in a set of equations that contain both dependent variables (that is, Ψ and ζ) clearly coupled. This facilitates discretisation. This method is described in detail by Pearson [36], and has been followed up by many authors including Campion-Renson and Crochet [37], Dhatt *et al* [38], and Peeters *et al* [39].

However, there are some major disadvantages to this method. The value of the vorticity at a wall can be difficult to specify, and the pressure field is not explicitly solved, which may be a desired quantity. Though there are methods to determine the pressure from the vorticity, these are computationally demanding and negate any savings made from solving the vorticity equation as opposed to solving the momentum and continuity equations. Furthermore, the technique cannot easily be extended to three dimensions, and is hence limited to only two-dimensional problems.

In three dimensions, a velocity-vorticity method may be employed, described by Guevre-

mont *et al* [40]. Again, the pressure does not need to be solved, but is expensive to obtain *a posteriori* if desired.

2.5.4 Babuska-Brezzi conditions

These are two conditions which impose restrictions on a solution scheme. Violation of either of these usually results in spurious pressure modes (see Zienkiewicz and Taylor [22]). The conditions state that:

- (i) the matrix operating on the pressure variables must be non-singular
- (ii) the divergence operator on the pressure must have no null states when the pressure is non-zero in the entire domain.

The first condition is met when the number of unknown nodal values of velocity is greater than the number of unknown nodal values of pressure. This condition is applicable when a coupled scheme is in use. The second condition ensures that only realistic pressure modes are present in the solution.

2.6 SIMPLE: A Segregated Approach

Coupled schemes, such as the primitive variable model have been very popular. The velocity and the pressure are calculated simultaneously from some suitable discretisation of the Navier-Stokes equations (see Reddy [23]). This means that for a problem with n degrees of freedom, a set of linear equations of size n must be solved, which can make solution time for problems with many elements (especially three dimensional problems) very long. Moreover, zeroes appear on the leading diagonal, thus requiring special attention to pivoting to obtain solutions. Furthermore, convergence will only occur if the interpolations for the variables satisfy the Babuska-Brezzi conditions (see Section 2.5.4),

necessitating different orders of elements for the velocity and the pressure. This effectively means that two meshes have to be built for a problem, one for velocity and one for pressure. Whilst this is not too problematic for structured domains, serious difficulties may be encountered with unstructured domains.

The term *segregated* is ambiguous, since it may be used to describe two very distinct approaches. One may either split a solution procedure into a series of sub-problems, which each have a specific mathematical character (for example, Glowinski and Pironneau [16]), or one may discretise the equations into a series of steps such that only a subset of the primitive variables are solved at any one time. The former approach is advantageous in that the mathematical properties of each problem may be exploited. However, in implicit formulations, this still necessitates the solution of linear equation systems of size n for problems with n degrees of freedom. Henceforth, the term *segregated* is used to describe the latter approach.

The approach of Shaw and others is interesting in that the algorithm is formulated so that the size of the linear equations system is reduced, by solving for each of the primitive variables separately. This results in the solution of several linear equation systems of size $n/4$ corresponding to each of the primitive variables. This is important, as the size of the global stiffness matrices produced are not proportional to the number of primitives, as is the case with coupled formulations. In implicit formulations this means a significant reduction in the storage required for the global stiffness matrix. The methodology takes ideas from FV schemes, in which estimates for the velocities are made using the momentum equation, and subsequently corrected using the modified form of the continuity equation to satisfy continuity at the end of each calculation. This is the SIMPLE approach and was devised by Patankar [24]. SIMPLE and its variants are discussed in two following sections, after which a segregated FE algorithms are considered.

2.6.1 The SIMPLE algorithm

One approach used in the FV method is that of Patankar [24]. The SIMPLE (Semi-Implicit Method for Pressure-Linked Equations) technique splits the velocity components into two parts. Each primitive variable ϕ (which may be either a velocity component or pressure), is split into two components:

$$\phi = \phi^* + \phi' \quad (24)$$

The starred values represent initial estimates of the variables; the dashed values are subsequent corrections. This splitting allows a solution procedure to be formed in which the pressure may be calculated: From a guessed initial pressure field, the velocity is calculated using the momentum equations to generate the values u^* , v^* , and w^* . Then a correction to the pressure may be found; the equation used to do this is called a *pressure correction equation*. The components of velocities calculated do not solve the continuity equation, thus the corrected values of pressure are used to calculate velocity corrections which are divergence free. At this point the velocities do not solve the momentum equations, and iteration is required to refine the primitives. As a solution converges, the values of the corrections reduce so that the primitives solve the governing equations. This important method is used to derive the governing equations for the scheme that is used in this work, therefore, a more detailed discussion is postponed until later.

The steps algorithm is given below:

1. Guess the pressure field p^* .
2. Solve the momentum equations, to obtain u^* , v^* , w^* .
3. Solve the p' equation.

4. Calculate p by the definition $p = p^* + p'$.
5. Calculate u', v', w' from their starred values using the velocity correction formulae.
6. Solve the discretisation equation for other variables (such as temperature and turbulence quantities) if they influence the flow field. If they do not influence the flow field, then their calculation is best left until after a converged solution for the flow field has been made.
7. Treat the corrected pressure p as the new guessed pressure p^* , and go to step 2, repeating the procedure until a converged solution has been found.

Note that during each iteration the velocities are updated by the velocity correction formulae, and hence solve the discretised continuity equation exactly. Thus, in a SIMPLE algorithm, the solution aims for a final solution by a series of continuity satisfying velocity fields. This feature of SIMPLE is important because the solution is less likely to diverge.

2.6.2 Variants of SIMPLE

The SIMPLE algorithm has been successfully used by the FV community. However, attempts to improve the convergence rate have led to SIMPLER which stands for SIMPLE Revised (see Patankar [24]). SIMPLER takes fewer iterations to converge because it calculates a pressure field (p^*) from *pseudo-velocities* (which are composed of the velocities of the nodes surrounding the node in question), before solving the momentum equations to find u^*, v^*, w^* . A pressure correction equation is still solved, but this is used only to update the velocities, and not the pressure field, which remains unchanged. The algorithm is shown below:

1. Start with a guessed velocity field

2. Calculate the coefficients for the momentum equations and hence calculate the pseudovelocities
3. Calculate the coefficients for the pressure equation and solve it to obtain the pressure field
4. Treating this pressure field as p^* , solve the momentum equations to obtain u^* , v^* , w^* .
5. Calculate the mass source and hence solve the p' equation.
6. Correct the velocity field using the velocity correction formulae, but *do not* correct the pressure
7. Solve the discretisation equations for other variables if necessary
8. Go back to step 2 until convergence.

The following points have been noted by Patankar [24].

1. In SIMPLE, the guessed pressure field is important. However, SIMPLER actually extracts a pressure field from a given velocity field, and does not use a guessed pressure.
2. If SIMPLER is given a correct velocity field, then the pressure equation will extract the correct pressure field automatically and no more iterations will be necessary. However, if the correct velocity field was given to SIMPLE, along with a guessed pressure field, then the solution would initially deteriorate because the guessed pressure would lead to starred velocities which would differ from the correct given ones.
3. Although SIMPLER converges in fewer iterations than SIMPLE, one iteration in SIMPLER requires more computational effort than one iteration in SIMPLE. This

is because SIMPLER has to solve a pressure equation in order to extract the pressure field, and because the pseudo-velocities have to be calculated. However, the fewer iterations required by SIMPLER lead to a faster solution than SIMPLE.

SIMPLE is a form of the pressure correction methods, where the time dependent momentum equations are solved along with a Poisson equation for the pressure, obtained by taking the divergence of the momentum equations and expressing the condition of the divergence-free velocity field (see Hirsch [14]). The SIMPLER modification, on the other hand is a form of the pressure projection method of Chorin [41]. In this method, the incompressibility constraint is indirectly satisfied by solving the Poisson equation for pressure in the primitive variable formulation. Another method which exists is pressure update algorithm. In this method the continuity equation is satisfied through penalising the discretised continuity equation on the right hand side of the discretised pressure equation. This is a recently developed scheme was investigated by Haroutunian *et al* [42], who showed that it did not perform as well as the pressure-projection method.

Many other variants of SIMPLE also exist. No comprehensive comparisons exist in the literature but various authors have made some comparisons:

The performance of five discretisation methods, SIMPLE, CTS-SIMPLE, SIMPLER, SIMPLEC and FIMOSE has been evaluated for three turbulent flows, by requiring a certain level of residual in mass, momentum and turbulent kinetic energy to be attained. It is found that SIMPLEC is the most efficient and stable method for solving the complex turbulent flows and its performance is not always identical to CTS-SIMPLE. SIMPLER needs the fewest iterations (see Chao and Ho [43]).

SIMPLEC's supremacy has been verified by Tamamidis and Assanis [44], who compared SIMPLE and SIMPLEC, and reports that SIMPLEC is found to produce computational time savings of up to 40% over SIMPLE, when used on problems of complex geometry

using a generalised body fitted co-ordinate control volume methodology for laminar flows.

The SIMPLE method is implicit, so iteration is required at each time step in order to converge the solution of the equations to a sufficiently accurate level. This can be time consuming. The PISO (Pressure implicit by splitting of operators) algorithm of Issa [45] avoids this by not iterating.

PISO, SIMPLER and SIMPLEC are compared by examining the computational effort required to obtain the same level of convergence in four test problems for steady flow. When the momentum equation is not coupled to a scalar variable the PISO algorithm is best, but when the scalar variable is strongly coupled to the momentum equation, SIMPLER and SIMPLEC exhibit better behaviour. In this case, PISO only produces reasonable solutions with small time steps. No clear superiority between SIMPLER and SIMPLEC was observed (see Jang *et al* [46])

Work on PISO is still progressing. For example, a new version called EPISO is an extension of PISO for motor engines, based on the three dimensional conservation equations governing the flow processes in asymmetric engine chambers, cast into FV form. It is at this point the PISO technique is applied - this solution procedure is an order of magnitude faster than those based on SIMPLE, as shown by Ahmadi-Befrui *et al* [47].

It is clear that there is some ambiguity over which scheme performs the best, in any particular situation, though most reports do indicate that SIMPLER is better than SIMPLE in terms of reduced computing time.

2.7 Segregated Finite Element Algorithms

Segregated FE schemes have been considered by many authors including Benim and Zinser [48], Rice and Schnipke [49], Haroutunian *et al* [42], Haroutunian *et al* [35] and Ramaswamy and Jue [34].

The schemes of Benim and Zinser [48] and of Rice and Schnipke [49] use SIMPLE techniques for segregation, but solved only steady-state problems. The scheme of Rice and Schnipke [49] uses bi-linear elements for both velocity and pressure, but requires boundary conditions containing mass flux terms to calculate the pressure. Such terms may be difficult to specify. Benim and Zinser [48] use different orders of accuracy for the velocity and pressure in order to satisfy the Babuska-Brezzi conditions and to avoid checkerboard solutions. This technique has the disadvantage of requiring two meshes to solve any one problem, and whilst this is not problematic for simple geometries, this can be restrictive in complex geometries.

Transient segregated methods have been investigated by Haroutunian *et al* [42] and Haroutunian *et al* [35]. The algorithms investigated were of pressure correction (PC) formulation, pressure projection (PP) formulation and pressure update (PU) formulation. In PC and PP algorithms are FE counterparts of the SIMPLE and SIMPLER algorithms respectively. Thus, in a PP formulation, the incompressibility constraint is indirectly satisfied by solving a Poisson equation for pressure. The PU method is slightly different as the continuity equation is satisfied through penalising the discretised continuity equation in the discretised pressure equation.

In each of these algorithms, direct Gaussian elimination was used to solve the linear equation systems. The PP was found to be the fastest algorithm, and this was compared to a fully coupled (FC) algorithm. It was found that while the PP algorithm requires substantially less storage than the FC algorithm, it performs faster only on very large (and physically complex) 2D problems and on most 3D problems. However, the handling of boundary condition terms in the PP algorithm is more complex than in the algorithm of Shaw [19], [20].

Ramaswamy and Jue [34], described the application of a second order accurate Taylor-Galerkin based segregated FE method, based on the PP method. However, whilst the segregation of the scheme allows equal-order interpolation for the velocity and pressure

components, the method is explicit and thus restricted by the CFL condition.

Shaw's algorithm [19], [20] is transient and involves ideas from both of these schemes resulting in a formulation which uses equal-order elements for both the velocity and pressure, allowing the same mesh to be used for both velocity and pressure. It is a SIMPLE (or pressure correction) type scheme and thus does not require the more complex boundary conditions of a SIMPLER type algorithm. A full description is given below.

2.8 Description of the Scheme

Consider the two-dimensional incompressible Navier-Stokes equations, for constant μ :

$$\rho \frac{\partial u}{\partial t} + \rho u \frac{\partial u}{\partial x} + \rho v \frac{\partial u}{\partial y} = -\frac{\partial p}{\partial x} + \mu \frac{\partial}{\partial x} \left[\frac{\partial u}{\partial x} \right] + \mu \frac{\partial}{\partial y} \left[\frac{\partial u}{\partial y} \right] \quad (25)$$

$$\rho \frac{\partial v}{\partial t} + \rho u \frac{\partial v}{\partial x} + \rho v \frac{\partial v}{\partial y} = -\frac{\partial p}{\partial y} + \mu \frac{\partial}{\partial x} \left[\frac{\partial v}{\partial x} \right] + \mu \frac{\partial}{\partial y} \left[\frac{\partial v}{\partial y} \right] \quad (26)$$

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \quad (27)$$

where u and v are the velocity components in the x - and y -directions respectively, p is the fluid static pressure, t is time, μ is the viscosity and ρ is the fluid density.

The method of Shaw [19], [20] discretises equations (25), (26) and (27) in the following manner. The time dependent terms in the momentum equations can be discretised using a first-order difference scheme, taking differences between values at the $(n+1)$ th and n th time level, for example, for a variable r

$$\frac{\partial r}{\partial t} = \frac{r^{n+1} - r^n}{\delta t} \quad (28)$$

where the time step, δt , is the time difference between time levels.

Using weighted average approximations, for the convection and diffusion coefficients and the pressure gradient, the x -momentum equation (25) becomes:

$$\begin{aligned} \rho \left[\frac{u^{n+1} - u^n}{\delta t} \right] + \rho \left[\theta \bar{u}^{n+1} \frac{\partial u^{n+1}}{\partial x} + (1 - \theta) \bar{u}^n \frac{\partial u^n}{\partial x} \right] + \rho \left[\theta \bar{v}^{n+1} \frac{\partial u^{n+1}}{\partial y} + (1 - \theta) \bar{v}^n \frac{\partial u^n}{\partial y} \right] = \\ -\theta \frac{\partial p^{n+1}}{\partial x} - (1 - \theta) \frac{\partial p^n}{\partial x} + \mu \frac{\partial}{\partial x} \left[\theta \frac{\partial u^{n+1}}{\partial x} + (1 - \theta) \frac{\partial u^n}{\partial x} \right] + \mu \frac{\partial}{\partial y} \left[\theta \frac{\partial u^{n+1}}{\partial y} + (1 - \theta) \frac{\partial u^n}{\partial y} \right] \end{aligned} \quad (29)$$

where θ is the weighting factor and a bar over a term denotes that that term is assumed constant for one iteration. Such terms are held constant to produce a linearised form of the equations.

This can be written with all the u -velocity terms at the $(n + 1)$ th time step on the left hand side, giving:

$$\begin{aligned} \rho \frac{u^{n+1}}{\delta t} + \rho \theta \bar{u}^{n+1} \frac{\partial u^{n+1}}{\partial x} + \rho \theta \bar{v}^{n+1} \frac{\partial u^{n+1}}{\partial y} - \mu \frac{\partial}{\partial x} \left[\theta \frac{\partial u^{n+1}}{\partial x} \right] - \mu \frac{\partial}{\partial y} \left[\theta \frac{\partial u^{n+1}}{\partial y} \right] = \\ -\theta \frac{\partial p^{n+1}}{\partial x} - (1 - \theta) \frac{\partial p^n}{\partial x} + \frac{\rho u^n}{\delta t} - \rho(1 - \theta) \bar{u}^n \frac{\partial u^n}{\partial x} - \rho(1 - \theta) \bar{v}^n \frac{\partial u^n}{\partial y} \\ + \mu \frac{\partial}{\partial x} \left[(1 - \theta) \frac{\partial u^n}{\partial x} \right] + \mu \frac{\partial}{\partial y} \left[(1 - \theta) \frac{\partial u^n}{\partial y} \right] \end{aligned} \quad (30)$$

The value of each variable at any point within an element can be described as:

$$\phi = \sum_{i=1}^n N_i \phi_i \quad (31)$$

where ϕ represents any one of the three variables u , v or p , and where n is the number of nodes on the element. Also ϕ_i is the value of ϕ at node i , and N_i is a function of position.

Using a Galerkin weighted residual method to produce the element equations, integrating by parts where necessary, results in the x -momentum equation becoming,

$$\begin{aligned}
& \int \left[\rho \frac{N_i N_j}{\delta t} + \rho \theta \bar{u}^{n+1} N_i \frac{\partial N_j}{\partial x} + \rho \theta \bar{v}^{n+1} N_i \frac{\partial N_j}{\partial y} + \mu \theta \frac{\partial N_i}{\partial x} \frac{\partial N_j}{\partial x} + \mu \theta \frac{\partial N_i}{\partial y} \frac{\partial N_j}{\partial y} \right] u_j^{n+1} d\Omega \\
& - \int N_i \left[\mu \theta \frac{\partial u^{n+1}}{\partial x} n_x + \mu \theta \frac{\partial u^{n+1}}{\partial y} n_y \right] d\Gamma = \\
& \quad \int (N_i \left[-\theta \frac{\partial p^{n+1}}{\partial x} - (1-\theta) \frac{\partial p^n}{\partial x} + \frac{\rho u^n}{\delta t} - \rho(1-\theta) \bar{u}^n \frac{\partial u^n}{\partial x} - \rho(1-\theta) \bar{v}^n \frac{\partial u^n}{\partial y} \right] \\
& - \frac{\partial N_i}{\partial x} \mu(1-\theta) \frac{\partial u^n}{\partial x} - \frac{\partial N_i}{\partial y} \mu(1-\theta) \frac{\partial u^n}{\partial y}) d\Omega + \int N_i \left[\mu(1-\theta) \frac{\partial u^n}{\partial x} n_x + \mu(1-\theta) \frac{\partial u^n}{\partial y} n_y \right] d\Gamma
\end{aligned} \tag{32}$$

where Ω denotes the domain, and Γ the boundary of the domain. The x - and y - components of a unit normal vector to the boundary are n_x and n_y respectively.

Looking at equation (32), the boundary terms are significant in that they define the boundary conditions for the discrete equation. The essential boundary condition is that u should be specified on the boundary, which will eliminate the equations at the boundary. The natural boundary condition is that the boundary terms in (32), components of the velocity gradient must be specified or they will naturally be zero. In most problems, either the velocity is known and so can be specified, or the velocity gradient is zero, implying a fully developed flow. Consequently the boundary terms in (32) can in most cases be ignored.

This equation can be written in matrix form, ignoring the boundary terms:

$$A u_j^{n+1} = B p_j^{n+1} + C p_j^n + D u_j^n \tag{33}$$

where

$$A = \int \left[\rho \frac{N_i N_j}{\delta t} + \rho \theta \bar{u}^{n+1} N_i \frac{\partial N_j}{\partial x} + \rho \theta \bar{v}^{n+1} N_i \frac{\partial N_j}{\partial y} + \mu \theta \frac{\partial N_i}{\partial x} \frac{\partial N_j}{\partial x} + \mu \theta \frac{\partial N_i}{\partial y} \frac{\partial N_j}{\partial y} \right] d\Omega \tag{34}$$

$$B = \int -\theta N_i \frac{\partial N_j}{\partial x} d\Omega \quad (35)$$

$$C = \int -(1 - \theta) N_i \frac{\partial N_j}{\partial x} d\Omega \quad (36)$$

$$D = \int \left(\rho \frac{N_i N_j}{\delta t} + \rho(1 - \theta) \bar{u}^n N_i \frac{\partial N_j}{\partial x} + \rho(1 - \theta) \bar{v}^n N_i \frac{\partial N_j}{\partial y} - \mu(1 - \theta) \frac{\partial N_i}{\partial x} \frac{\partial N_j}{\partial x} - \mu(1 - \theta) \frac{\partial N_i}{\partial y} \frac{\partial N_j}{\partial y} \right) d\Omega \quad (37)$$

and a similar expression can be derived from the y - momentum equation.

From these discrete forms of the momentum equations, the value of the velocity components can be found, given an initial velocity and pressure field, and so the continuity equation must be used to calculate the pressure. This is done by splitting the variables into two parts: the values that satisfy the momentum equations (asterisked) and the corrections that ensure that continuity is satisfied (primed), i.e.

$$u = u^* + u' \quad (38)$$

$$v = v^* + v' \quad (39)$$

$$p = p^* + p' \quad (40)$$

As the velocity field at the n th time level satisfies the continuity equation, we ensure that the continuity equation is also satisfied at the $(n + 1)$ th time level, i.e.

$$\frac{\partial u^{n+1}}{\partial x} + \frac{\partial v^{n+1}}{\partial y} = 0 \quad (41)$$

hence using equations (38) and (39),

$$\frac{\partial u'^{n+1}}{\partial x} + \frac{\partial v'^{n+1}}{\partial y} = -\frac{\partial u^{*n+1}}{\partial x} - \frac{\partial v^{*n+1}}{\partial y} \quad (42)$$

By applying a Galerkin weighted residual method, this becomes, in discrete form:

$$\int N_i \left[\frac{\partial u'^{n+1}}{\partial x} + \frac{\partial v'^{n+1}}{\partial y} \right] d\Omega = - \int N_i \left[\frac{\partial u^{*n+1}}{\partial x} + \frac{\partial v^{*n+1}}{\partial y} \right] d\Omega \quad (43)$$

Integrating the left hand side by parts:

$$\begin{aligned} \int \left[-\frac{\partial N_i}{\partial x} N_j u_j'^{n+1} - \frac{\partial N_i}{\partial y} N_j v_j'^{n+1} \right] d\Omega + \int N_i \left[u'^{n+1} n_x + v'^{n+1} n_y \right] d\Gamma = \\ - \int N_i \left[\frac{\partial u^{*n+1}}{\partial x} + \frac{\partial v^{*n+1}}{\partial y} \right] d\Omega \end{aligned} \quad (44)$$

The boundary terms of this equation again define the boundary conditions required for solution. If the correction values for the velocity components are specified on the boundary, which is the essential boundary condition, the equations at the boundary are eliminated, and so the boundary terms are not needed there. However, on other boundaries, the correction velocities must be specified, as this is the natural boundary condition., if they are not to be zero automatically. When the velocity is known on the boundary, the correction values for the velocity components will be zero, but on the boundaries where the velocity components are not known, the corrections velocities are not known either. It is assumed here, that these corrections are small, which they will be if the solution converges during a given time step, and so the boundary integral in equation (44) can be ignored.

When the boundary term is ignored, the equation becomes

$$\left[\int -\frac{\partial N_i}{\partial x} N_j d\Omega \right] u_j'^{n+1} + \left[\int -\frac{\partial N_i}{\partial y} N_j d\Omega \right] v_j'^{n+1} = - \int N_i \left[\frac{\partial u^{*n+1}}{\partial x} + \frac{\partial v^{*n+1}}{\partial y} \right] d\Omega \quad (45)$$

The matrix form of the momentum equations can be rewritten in terms of the asterisked and primed values as:

$$A(u^{*n+1} + u'^{n+1})_j = B(p^{*n+1} + p'^{n+1})_j + Cp_j^n + Du_j^n \quad (46)$$

However, when the momentum equation is solved using equation (46), the values of u and v do not satisfy the continuity equation, and so the effective form of equation (46) for the calculation of u^* is:

$$A(u^{*n+1})_j = Bp_j^{*n+1} + Cp_j^n + Du_j^n \quad (47)$$

This can be subtracted from the full form to give:

$$Au_j'^{n+1} = Bp_j'^{n+1} \quad (48)$$

Therefore, the nodal values of the u -velocity can be found from:

$$u_j'^{n+1} = A^{-1} Bp_j'^{n+1} \quad (49)$$

and similarly, the nodal values of the v -velocity can be found from:

$$v_j'^{n+1} = A^{-1} Ep_j'^{n+1} \quad (50)$$

where E is the equivalent form of B for a y -derivative.

Using the above equations together with the discrete continuity equation, an equation for the pressure correction values can be found:

$$\left[\int -\frac{\partial N_i}{\partial x} N_j d\Omega \right] \frac{B}{a_{ii}} p_j'^{n+1} + \left[\int -\frac{\partial N_i}{\partial y} N_j d\Omega \right] \frac{E}{a_{ii}} p_j'^{n+1} = - \int N_i \left[\frac{\partial u^{*n+1}}{\partial x} + \frac{\partial v^{*n+1}}{\partial y} \right] d\Omega \quad (51)$$

where A^{-1} has been approximated by the reciprocal of the diagonal of A, that is $1/a_{ii}$.

Extending this to three dimensions is relatively straightforward.

2.9 Overview of Algorithm

For the two-dimensional case, the iterative procedure developed is as follows (see Shaw [19] and Shaw [20]):

1. Set the initial values at time step $n = 0$ of u^n, v^n and p^n to correspond to the known or assumed initial velocities and pressure fields.
2. So that the scheme has reasonable first guesses of velocity and pressure, assume that the values of the velocity and pressure at the $n + 1$ 'th time level can be given by the values at the n 'th time level.
3. Begin the internal iteration procedure within each time step by solving the discrete momentum equations, to give predicted values of the velocity components u^{*n+1} and v^{*n+1} .
4. From the discrete pressure correction form of the continuity equation extract the pressure field.
5. From the equations find the correction velocity components u'^{n+1} and v'^{n+1} .
6. Calculate the values of velocity and pressure that would satisfy the continuity equation, u^{n+1}, v^{n+1} and p^{n+1} .

7. As the values calculated in step (6) do not satisfy the momentum equations, it may be necessary to loop back to step (3) and perform more iterations for this time step. However, if the values of u^{n+1} , v^{n+1} and p^{n+1} have converged, the number of iterations can be suspended, and the calculation for the next time step may begin.
8. Update the values of u^n , v^n and p^n , and begin a new set of internal iterations at step (3) for the next time step.

Essentially the scheme is a SIMPLE Galerkin FE method in which each of the primitive variables are solved without needing to solve more than one variable at a time. This contrasts with primitive variable FE schemes. The scheme calculates for the pressure using a pressure correction form of the continuity equation, thereby automatically satisfying the first Babuska-Brezzi condition since the matrix used is not singular. It is necessary however, to use an approximation for the inverse of the global stiffness matrix as opposed to the exact inverse otherwise singularities would occur. The approximation used is simply the inverse of the leading diagonal; other approximations may be valid but these have not been explored. This is a direct result of a segregated formulation and is not necessarily true with a coupled formulation.

The second condition is satisfied since the pressure correction values are formed from a Laplacian-like operator. This implies that the solutions for pressure will be smooth and oscillatory modes never form (see Shaw [20]). Thus equal-order interpolation can be used for both the velocity and pressure. This is highly desirable, since it means that once a mesh has been generated, the same nodes can be used for both velocity and pressure. The scheme is first-order accurate in time (see Shaw [19]) and uses tri-linear (hexahedral) elements.

Almost all of the non-scalar data may be held in simple arrays, with the exception of the global stiffness matrix. The FE method produces large sparse matrices which are banded (see Reddy [23]), and so some kind of storage scheme is needed to hold the non-zero data as efficiently as possible. Moreover, discretisation of the convection term results

in non-symmetric matrices, thus skyline storage is used, where three arrays are defined. One stores the leading diagonal, whereas the other two are large arrays used to store the upper and lower parts of the matrix. One further array is necessary to indicate the starting positions of each row/column of the matrix in the two large arrays (the matrix is symmetric in shape). Skyline storage is discussed in detail by Dhatt and Touzot [50] and also in Griffiths and Smith [51].

2.10 Input Data and Parameter Importance

The solution of any given problem using the implementation of Shaw's algorithm [19], [20] requires three main types of data. These describe the characteristics and geometry of the problem, discretisation data, and the solution scheme which identifies how the flow should be calculated. These are summarised in Table 1 and discussed in the following sections.

2.10.1 Primary data

Whilst the viscosity and density of the fluid under simulation is determined by the problem itself, the choice of initial and boundary conditions must be chosen with care. The problem domain must be selected so that all areas of interest are captured such that the boundary conditions can be well defined, and yet must be of reasonable enough size to allow a meaningful resolution of the domain. Sometimes it is possible to use the symmetry of a situation in order to reduce the domain size. For example, in the analysis of viscous flows in a two dimensional pipe at low Reynolds number, it is sufficient to model half of the pipe, imposing the centreline velocity on one of the edge of the pipe which is parallel to the direction of flow, and the no-slip condition on the opposing edge. However, this assumption implies that the flow itself is symmetric, and this may not always be the case, even when geometrical symmetry exists. A classic example of this is the flow around a circular cylinder, which becomes asymmetric once the Reynolds number exceeds 40 (see

Tritton [2]). Ideally the initial conditions should be as close to a realistic solution as possible. This can be tricky (especially for transient problems), but experience has shown that it is easier to obtain convergence on a coarse mesh than on a finer one. This has suggested the following strategy for a problem that initially diverges: run the problem on a coarser mesh, obtain a converged solution and then project the solution onto the original mesh and run the problem again in order to capture the finer details of the flow.

Prescription of boundary conditions is non-trivial. For velocity, the Dirichlet conditions for velocity are easy to enforce, though one has to be wary of singularities which may occur. For example, in the analysis of a driven cavity problem (where the domain is square or cuboidal, with one of the walls moving at constant velocity while the others are stationary), singularities occur at the corners of the domain which may corrupt the solution. Numerically, this means that just one element or cell has to account for the discontinuity from the static wall to the moving wall. However, this may be dealt with in different ways. One can either set the velocity of the node next to the node of zero velocity at half the speed of the moving wall, or a mathematical function can be used to describe the speed of the moving wall which has no singularities at the corners.

Pressure boundary conditions also require caution. Since pressure is relative, it is always essential to specify the pressure at least one point in the mesh, otherwise the matrices generated from the pressure correction equation will be singular. For example, at the outflow of a pipe, it is common to simply set the pressure to zero. This can have an upstream effect and lead to divergence of the solution, unless substantial relaxation is imposed on the pressure. Moreover, points where pressure has been imposed may appear to act as sources or sinks during the solution process. This permits a necessary condition for convergence - such points should not be sources or sinks. Other discretisations (for example, Gresho *et al* [52]) integrate the pressure gradient term by parts. This allows less restrictive pressure boundary conditions to be imposed. However, modifications to the original algorithm to allow such boundary conditions are beyond the scope of this work.

2.10.2 Secondary data

Appropriate discretisation of the domain is crucial in obtaining physical meaningful flow solutions. The number and order of elements should be such that the problem is sufficiently resolved without compromising storage and runtime. Thus one must ensure that in regions of steep gradients, such as close to inlets and boundaries, an adequate number of elements is used, and that elements are not 'wasted' in regions of low gradients. It is possible for a solution to be convergent on one mesh, with residuals dropping to zero, even though the solution is incorrect due to a mesh which does not have fine enough resolution. In such a situation, aspects of the flow may be completely hidden from the solution generated. Thus one simple check for mesh convergence is to run a problem on a finer mesh, and check that the solution has not changed from one mesh to another.

2.10.3 Tertiary data

Time step:

Once the mesh has been defined, the next step is to specify the nature of the solution scheme. For steady-state problems, it is normal merely to set the time step to a large value (in the order of 10^4 or more). This works for simple problems. However, for anything more complex the scheme may diverge. This is due to the stabilising effect that the transient terms have on the solution. A large time step reduces these values and can destabilise the solution, especially if starting from unrealistic conditions. Therefore, the solution of steady-state problems may require a transient solution, until a more realistic solution is produced. Thereafter the time step may be set to a higher value, with a greater likelihood of convergence.

For transient problems, it is desirable to have the time step as large as possible in order to reduce computation time. However, the time step must be small enough to:

- (i) capture transient phenomena accurately

(ii) prevent divergence due to the decreasing stabilisation of large time steps.

One useful concept in determining the actual value for the time step, is the idea of the residence time (t_{res}) of an arbitrary fluid particle within a representative element (see Shaw [28]). This is defined below:

$$t_{res} = \frac{\delta x}{u} \quad (52)$$

where δx is a characteristic length of the element, and u is the velocity.

This defines an upper limit for the time step, since any fluid particle should stay within an element for at least one time step, unless further stabilising is applied (for example, upwinding). A conservative choice is to set the time step to half of the value of the residence time (see Shaw [28]).

Time scheme:

For first-order approximations of the time derivative, the dependent variables at two consecutive time steps may be weighted in the following manner:

$$\theta \frac{\partial u_{n+1}}{\partial t} + (1 - \theta) \frac{\partial u_n}{\partial t} = \frac{u_{n+1} - u_n}{\delta t_{n+1}} \quad (53)$$

where δt_{n+1} is the time step, u_n refers to the value of the variable at time step n , and θ is the weighting factor.

By choosing different values of θ , well-known difference schemes may be selected as shown below:

1. $\theta = 0$: Forward-Difference (Euler)

2. $\theta = \frac{1}{2}$: Crank-Nicholson
3. $\theta = \frac{2}{3}$: Galerkin
4. $\theta = 1$: Backward-Difference

The forward difference method is stable only for sufficiently small time steps whereas the Crank-Nicholson scheme is usually stable in time for large time steps. The backward difference and Galerkin methods are stable for arbitrarily large time steps. Moreover, the forward difference scheme results in a fully explicit process with the resulting global stiffness matrix being diagonal, whereas the backward difference results in a fully implicit scheme. The Crank-Nicholson and Galerkin methods are semi-implicit (see Press *et al* [21]).

The forward and backward difference methods are first-order accurate in time, unlike the other time-schemes, which are second order accurate. Therefore, when it is important to capture transient phenomena, a second-order time accurate scheme is desirable. However, it has been noted that the unconditionally stable schemes may take excessively long to converge suggesting the use of the first-order, fully implicit backward differences. An alternative is to perturb the weighting value by a value less than $O[(\delta t)^2]$ in order to retain second order accuracy whilst increasing the speed of convergence due to the time scheme (see Fletcher [53]).

Relaxation Parameters:

At the end of each time step, new values are calculated for the dependent variable. However, convergence can be controlled by relaxing the new values of the variables with the old values in the following manner:

$$u_{n+1} = \theta_u u_{n+1} + (1 - \theta_u) u_n \quad (54)$$

where θ_u is the relaxation parameter for velocity.

Relaxation is essential because the newly calculated values may be very different to the previously calculated values, and can diverge. The relaxation provides a control over a solution which is hopefully converging by not letting the solution change too much.

Relaxation on the pressure is of a slightly different form, in order to enforce continuity:

$$p_{n+1} = \theta_p p' + p_n \quad (55)$$

where p' is the extracted pressure from the pressure correction equation, and θ_p is the relaxation parameter for pressure.

However, problems have been found in getting the pressure to converge, and so an additional relaxation of the same form as the velocity has been tried:

$$p_{n+1} = \theta_p p' + (1 - \theta_p) p_n \quad (56)$$

This can have dramatic improvements on the convergence of the pressure, and leads to a converged accurate velocity field very quickly (often within only five non-linear iterations). However, the addition of this extra relaxation implies that continuity is not correctly enforced leading to an inaccurate pressure field. But if at this point the extra relaxation is removed, then the scheme has an accurate velocity field and a pressure field which is qualitatively correct from which to work. This has resulted in a pressure relaxation switch of the form:

$$p_{n+1} = \theta_p p' + (1 - \delta\theta_p) p_n \quad (57)$$

where δ is either 1 or 0.

2.11 Simple Test Cases and Code Validation

2.11.1 Validating the steady-state and time dependent terms

The first reported implementation solved the steady-state Navier-Stokes equations and was validated for a number of problems in (see Shaw [20]). Once the time-dependent terms were added, a Couette-Poiseuille flow problem was solved. The results were compared to the solution produced by a simple explicit finite difference scheme, with both schemes using an extremely small time step (0.0001sec). Very good comparison between the results was found. Moreover, the solution (which is steady-state, even though the algorithm ran in a pseudo-transient manner) has exact solutions, even though the program solves equation (58).

$$\rho \frac{\partial u}{\partial t} = -\frac{\partial p}{\partial x} + \frac{\partial}{\partial y} \left(\mu \frac{\partial u}{\partial y} \right) \quad (58)$$

The solution to this (for a fixed pressure gradient) is simply:

$$u = \frac{ky}{2\mu}(y - h) + \frac{cy}{h} \quad (59)$$

where $k = \frac{\partial p}{\partial x}$, c is the speed of the moving wall, and h is the separation of the two walls.

The parabolic solutions at the pressure gradients specified have the profiles given by equation (59).

2.11.2 Establishing the directional integrity of solutions

In order to test that the solution of the equations has no directional bias, a simple problem was rotated in space by non-zero angles. Solutions produced were then compared to the solutions generated for the original problem. In this problem, the entire problem is rotated, including the mesh geometry. Hence, both before and after rotation, the flow direction is parallel to many of the lines in the mesh.

Flow in a pipe is simulated using a coarse mesh, consisting of 36 nodes and 10 elements, representing half the width of the pipe. The flow is laminar and in one dimension, with Reynolds number based on the pipe diameter and maximum velocity equalling 0.018. The fluid has unit viscosity and density. The residence time given by equation (52) is 0.56 seconds, therefore a time step of 0.25 seconds is used, and the solution is run fully implicit. Relaxation parameters are set to 0.5 for velocity and 0.1 for pressure. Since the flow is governed by viscous effects, there is no need to use any extra relaxation on the pressure as described in Section 2.10.3. Parabolic flow is imposed at the inlet, and the pressure is set to zero at the outlet.

The flow is first calculated in the flow direction sense, and then the entire problem was rotated by a given set of angles (which were 30, 40 and 50 degrees). Modifications to the boundary conditions were of course necessary.

Both flows were calculated over 40 time steps with 5 non-linear iterations in each time step. Convergence was extremely fast, on this admittedly simple problem, and the pressure increment fell by 6 orders of magnitude over the solution time, in both cases. The domain of the problem and the velocity field may be seen in Figure 3.

Figures 4 and 5 give the percentage error from the analytical solution as the solution procedure progresses, for both the natural and the rotated system. We can see that for the first five time steps, the error for the rotated system is very slightly higher than for

the natural system. Between time steps 6 and 13 however, the natural system is slightly more inaccurate. On the whole however, the differences in solutions are slight and are probably attributable more to the round-off errors associated in the calculation of the error term than to any difference in the solution scheme.

2.12 Solution Strategies

Assuming that the primary and secondary data for the problem have already been defined, only the tertiary data remains to be set.

We have avoided using the explicit form for the time integration due to the restrictive time step this requires because of the CFL condition. Instead we have explored the other time schemes. On the simple parabolic flow test case it has been found that convergence can be extremely slow unless the scheme is fully implicit. Thus we have continued to use the fully implicit scheme.

The choice of the time step has been mainly a trial and error selection, from the upper bound defined by the residence time, given by equation (52). If convergence problems occur from an unrealistic solution, two strategies have been put into practice:

- (i) start from a potential flow solution,
- (ii) and/or run a few non-linear iterations per time step initially for many time steps, and then increase the number of non-linear iterations per time step.

For steady-state problems, it is usual to try running with a high value of time step (in the order of 10^4 or higher), but if this does not work, one of the following may be tried:

- (i) run the problem as if it were transient, with time steps comparable to the residence time until convergence over time is observed. At this point the solution is effectively

steady-state. This is known as time marching (see Hirsch [6]).

(ii) run the problem initially as pseudo-transient, and then switch to the original time step.

Primary data (the problem)	initial conditions, boundary conditions, viscosity and density
Secondary data (discretisation data)	choice of element type positioning of elements
Tertiary data (solution scheme)	choice of time step, relaxation parameters, number of non-linear iterations, type of time integration method

Table 1: Types of data required by the implementation of Shaw [19,20] to solve a flow problem. These are discussed in Section 2.10.

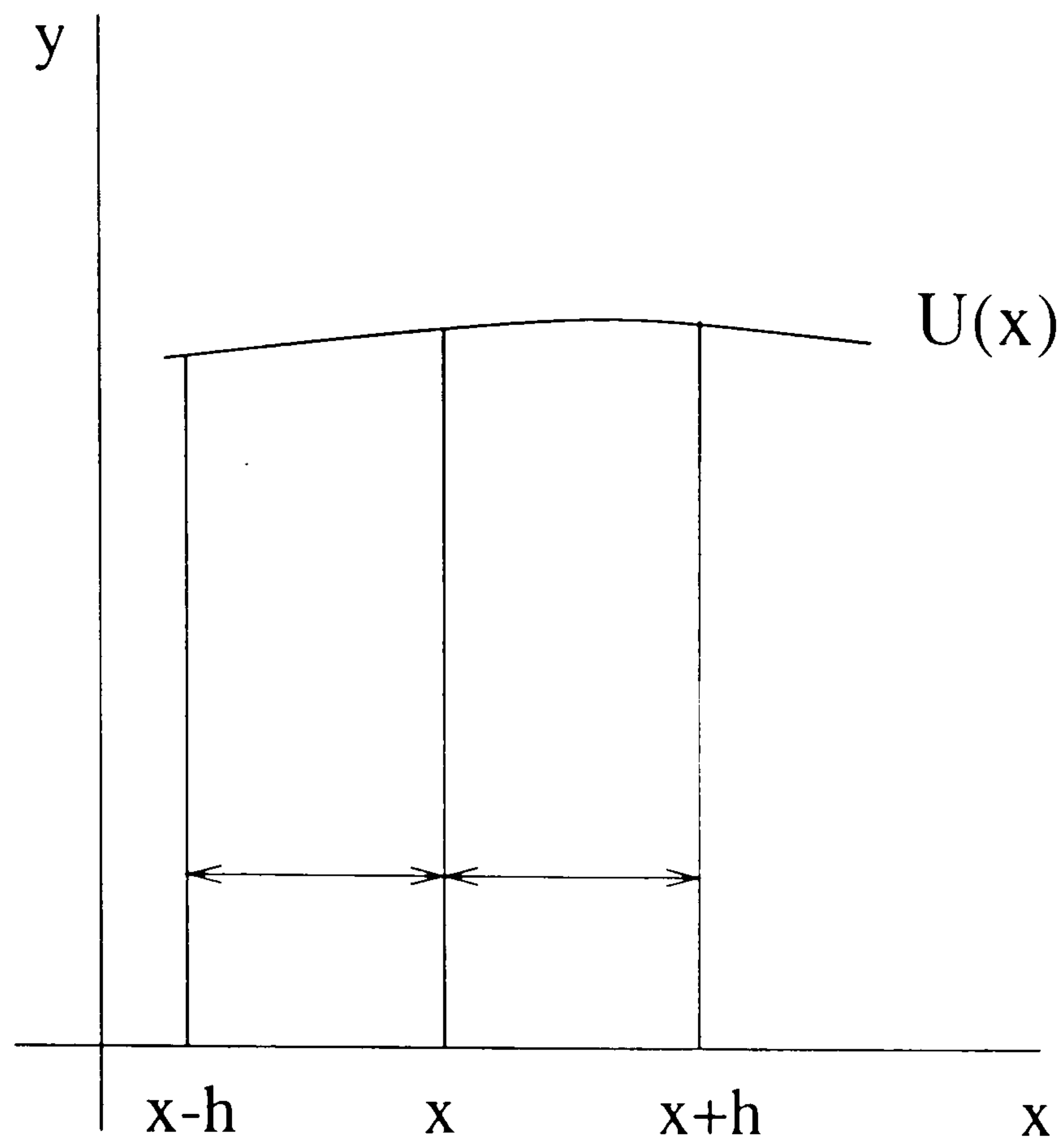


Figure 1: Central differencing of a first derivative of a continuous function. This is discussed in Section 2.2.1.

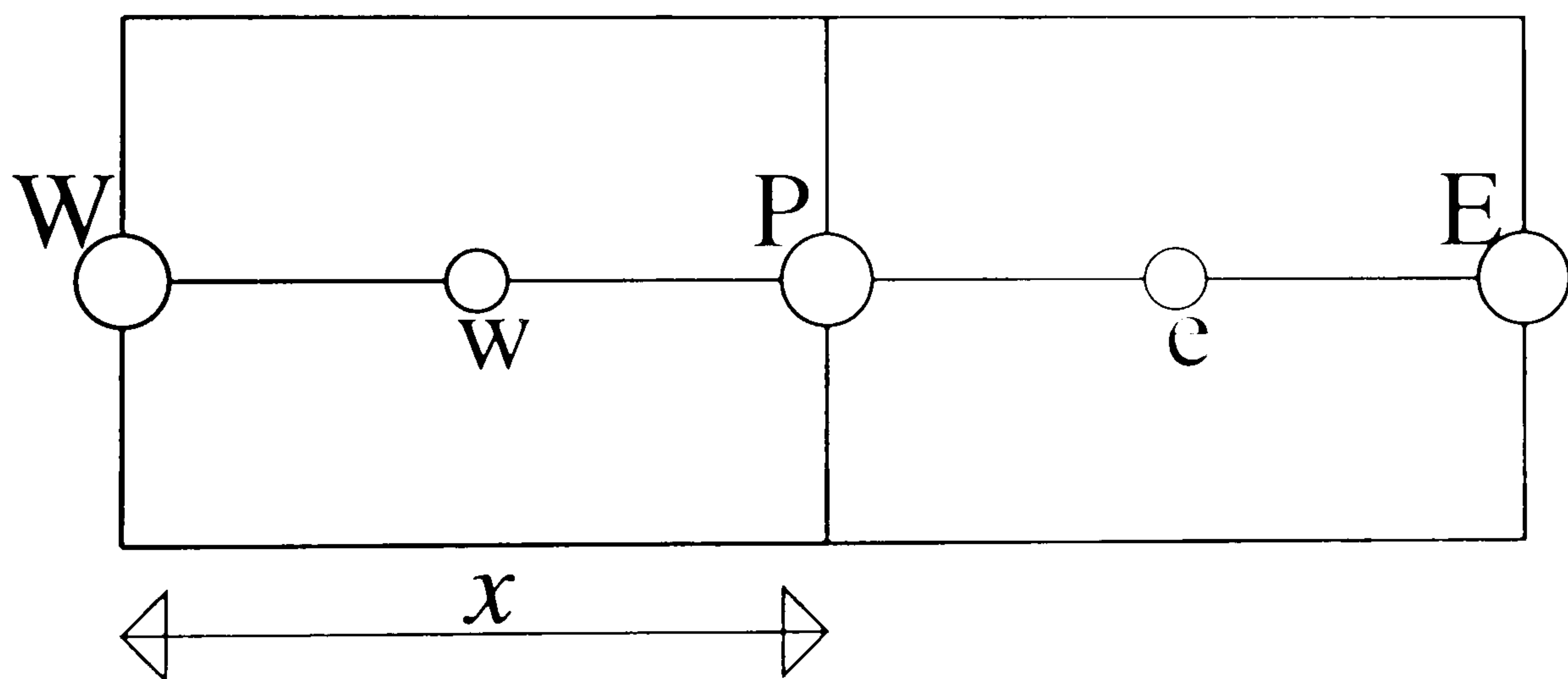


Figure 2: A simple control volume with two cells. Calculation of pressure values at P using central differences may lead to a checkerboard pressure field. This is discussed in Section 2.4.5.

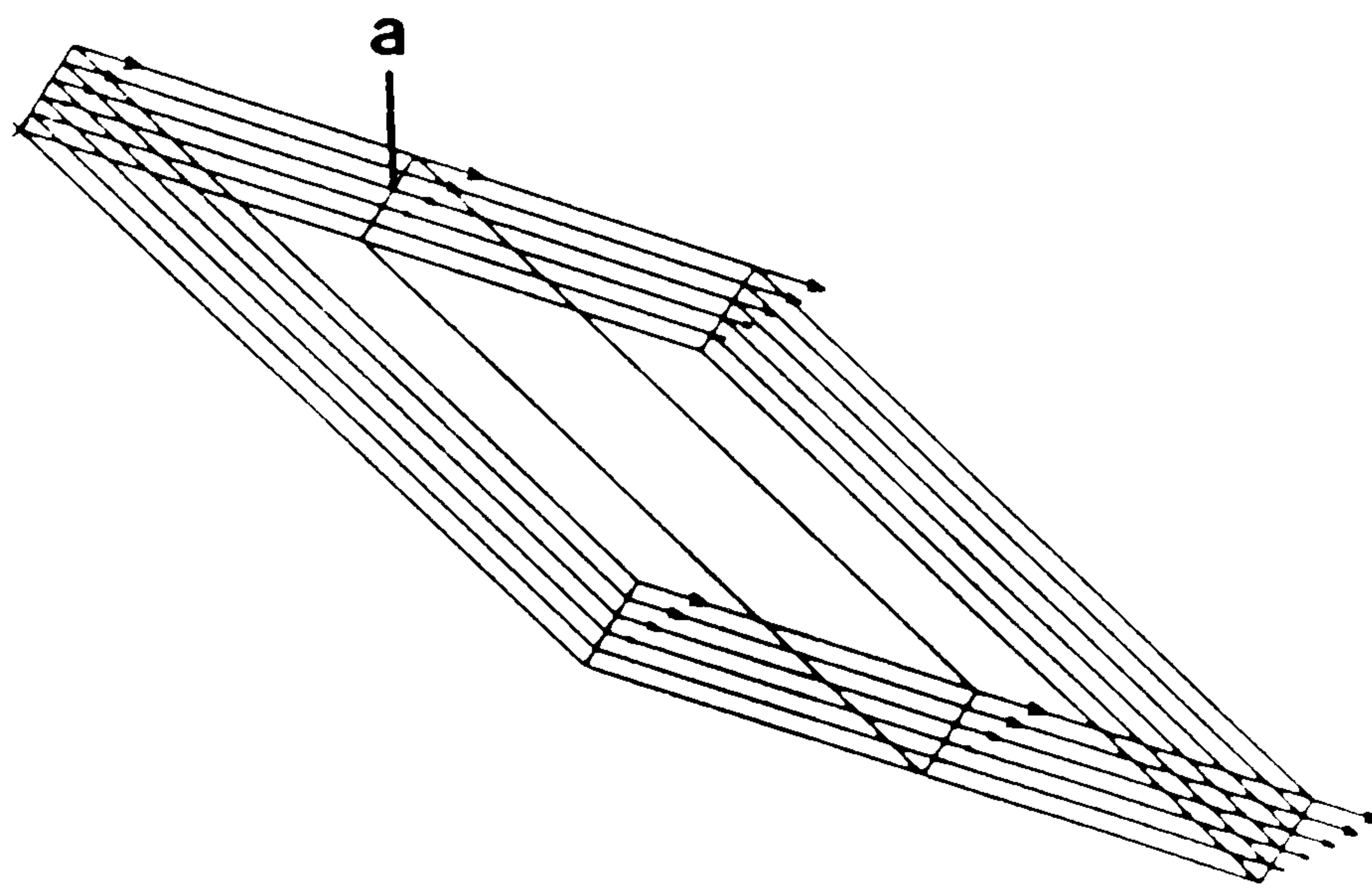


Figure 3: Velocity field for simple flow. The entire problem has been rotated in space in order to check that the each of the momentum equations interact properly to produce a proper solution. The point *a* is the monitor location. This experiment is further discussed in Section 2.11.2. The scale used is 5 mm : 1m/s.

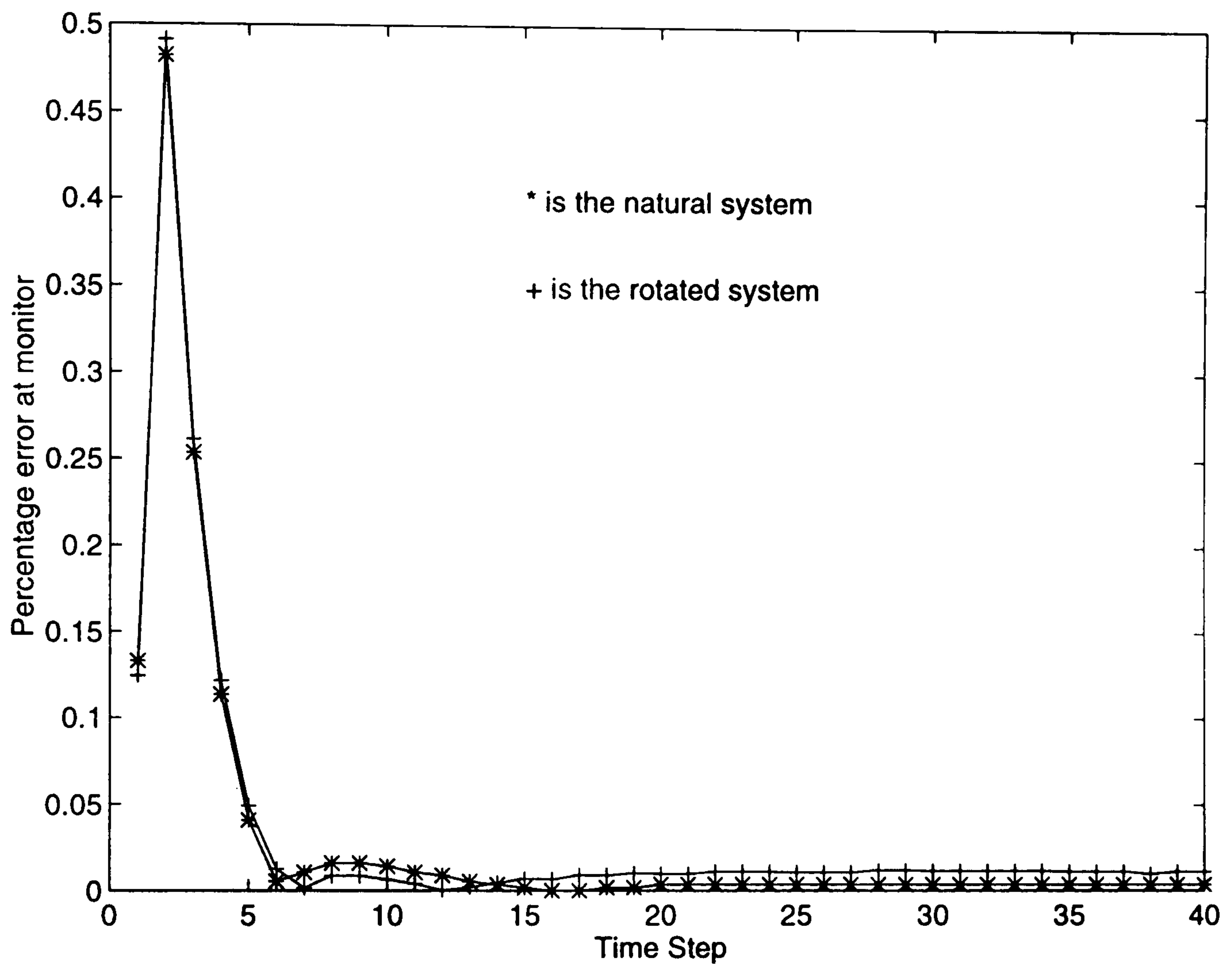


Figure 4: Effect of rotation of a complete flow problem on the values of velocity at one point in the flow. The experiment detailed in Section 2.11.2 considers the effect of rotating an entire problem in space, on the resulting solution. Percentage errors for the monitor velocity from both the original one dimensional problem and the rotated (and hence initially three dimensional problem) are shown compared to an analytical value, at the monitor, located at point *a* of the mesh shown in Figure 3.

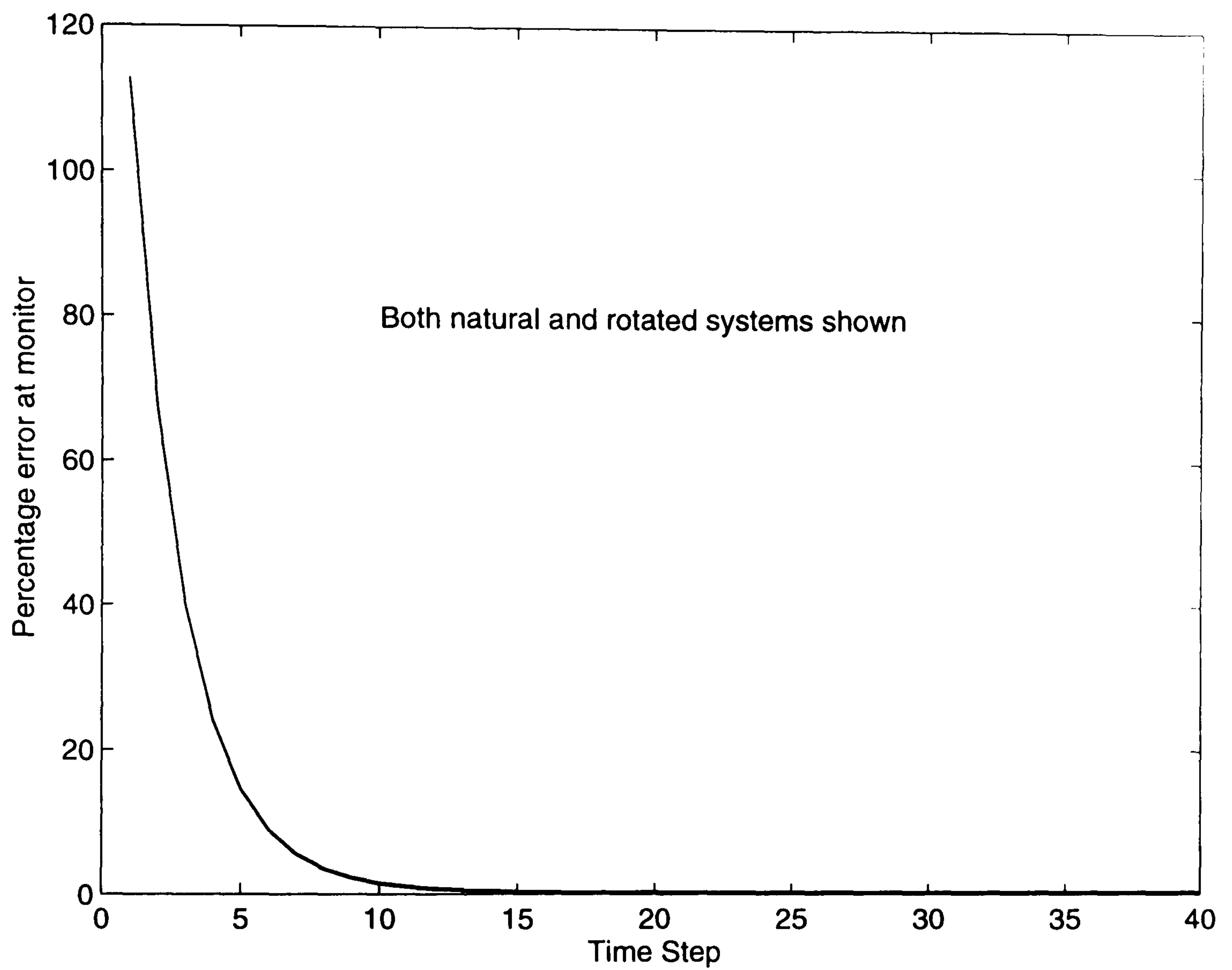


Figure 5: Effect of rotation of a complete flow problem on the values of pressure at one point in the flow. In this figure, the percentage pressure error is shown as opposed to the velocity monitor error described by Figure 4. The experiment itself is discussed in Section 2.11.2.

3 ACHIEVING FAST SOLUTIONS

There are three basic ways of reducing the computation time of an algorithm. One method is to reduce the computational overheads that are incurred by the program, without actually changing the algorithm in any way. This differs from algorithmic modifications which may be used to produce solutions by using cheaper techniques. Such techniques include mass lumping and iterative solvers. Alternatively, a faster machine may be used, for example a more powerful sequential computer or a computer with a different architecture, a subject discussed in Chapters 4 and 5.

Application of methods for reducing computational overheads can reduce computational time quite simply but may be limited in their success. Often, algorithmic modifications are inevitable to reduce CPU time, but these may have implications on the accuracy of the final solution. Therefore, any such modifications need to be explored.

3.1 Case Studies

Two case studies will be considered in the following sections. Case 1 consists of a set of problems which represent three dimensional cavity flow. Each problem consists of n^3 elements, where n is the number of elements on one side. Flow, mesh and domain considerations are not relevant, since this series of problems serves to exhibit timing characteristics of the solution procedure over problems of varying degrees of freedom. Details are shown in Table 2. It can be seen that the size of the skyline matrices required to store the global stiffness matrices generated rapidly increases with n .

Case 2 considers the flow which develops when uniform flow meets two infinitely long parallel plates. Analytical solutions for the fully developed flow exist, and thus allow a measure of the accuracy of a particular method. The plates are separated by one unit, and the domain is 15 units in the streamwise direction. Meshes considered are biased so that the elements adjacent to the inlet are a twentieth the size of the elements adjacent to

the outlet in the streamwise direction. Moreover, the elements are biased in the spanwise direction such that the velocity gradient (of the analytical solution) between spanwise nodes is constant. Details of the meshes used are given in Table 3. No slip conditions are enforced on each of the plates, and the inlet flow is a unit velocity profile. Pressure at the outlet is specified to zero to fix the pressure solution.

3.2 Profile of the Solver

Various UNIX utilities may be used to determine profiling characteristics of the code running any specific problem. These are *tcov*, which generates a number count for individual commands, *prof*, which produces a profile of the time spent in each subroutine, and *gprof*, which determines cycles across subroutine calls. However, these profiling tools require unoptimised code in order to function. Of these, *gprof* has proved to be the most informative, as it provides the most extensive information of the three tools.

Using the first case of problems described above the percentage times for the calculation and assembly of the element matrices, and the solutions of the linear equation systems is shown in Figure 6. This timing data was obtained on a Sparc Station 10.

It can be seen that initially the time required for the calculation and assembly of the element matrices dominate the execution time. However, as the problem size increases, the time for solution of linear systems of equations dominates. Together, the costs of these two processes occupy the solution time almost exclusively. We consider various ways of reducing these costs in the following sections.

3.3 Reducing Computational Overheads

The routine to solve the linear equation systems is an efficient LU decomposition solver, and is not amenable to further optimisation. Thus, reduction of computational overheads which occur exclusively in the calculation and assembly of the element equations can only

be effective for small problems.

Computational overheads occur in a variety of guises. These include, unnecessary variable initialisation, common sub-expressions repeatedly being evaluated, and the manner in which conditionals are evaluated. These are considered in the section below. One further overhead may arise from the way in which data is accessed from memory, as a compromise between the cost of accessing data and the total storage requirement must be made, especially when values are repeatedly extracted from a storage system. However, a discussion of this is postponed to Section 3.9 as it is more relevant within the context of iterative solvers.

3.3.1 Arithmetical rearrangement

Though each computer performs different algebraic operations in different ways dependent on the compiler, it is possible to estimate relative execution times for basic operations. Fletcher [53] presents results for several machines based on a documented program, intended to run on FORTRAN-77 with an unoptimised compiler under UNIX. However, from the point of view of numerically intensive codes, the program and resulting data presented is limited in that detailed analysis of the times for variables stored in array form is not considered. Since most intensive calculations will involve mainly array variables, this program has been extended, and results are shown in the Table 4.

In this table, FL refers to floating point operations. Values are times in seconds required to perform 10 million operations in single precision when the machines in question are used in single user mode. Machine dedication for all times is 99% or greater.

From this table we can see that operations involving addition, subtraction and multiplication typically take the same time, with the division operation taking longer. This is a trend also identified by Fletcher [53]. Array operations take about 30% longer than their scalar counterparts, and IF statements take almost as much time as scalar operations.

Under the assumption that each operation in an expression takes a finite amount of time, comparable to the values given in the above table, then one simple approach for reduction in computational overheads is

1. Factoring expressions into as short arithmetical form as possible,
2. Avoiding the use of division in intensive parts calculation, and instead pre-invert variables and multiply,
3. Using temporary variables for common sub-expressions to reduce the total number of operators that must be evaluated.

For example, in the following code fragment below, which is used to calculate the left hand side element stiffness matrices, 34 floating point operations must be evaluated.

```
elhs(i,j)= elhs(i,j)+const*(
    dt*rho*theta*unp1*sf(i)*gdsf(1,j)
    +dt*rho*theta*vnp1*sf(i)*gdsf(2,j)
    +dt*rho*theta*wnp1*sf(i)*gdsf(3,j)
    +dt*theta*amu*gdsf(1,i)*gdsf(1,j)
    +dt*theta*amu*gdsf(2,i)*gdsf(2,j)
    +dt*theta*amu*gdsf(3,i)*gdsf(3,j)) ,
```

This could be replaced by the following equivalent calculation which has 18 operators.

```
elhs(i,j)= elhs(i,j)+ const*dt*theta*(
    sf(i)*rho*(
    unp1*gdsf(1,j) + vnp1*gdsf(2,j) + wnp1*gdsf(3,j) ) +
    amu*( gdsf(1,i)*gdsf(1,j) +gdsf(2,i)*gdsf(2,j) +
    gdsf(3,i)*gdsf(3,j) ) )
```


One clear result is a loss in readability.

3.3.2 Common sub-expression evaluation

Consider the following code fragment,

```
C=A+B+D
```

```
E=Q+A+B
```

$A+B$ is a common sub-expression, and some computational work could be avoided if temporary variables are used. Compilers will sometimes do this, but never over subroutine calls, as the compiler cannot predict the changes in individual variables over subroutine calls.

Once these alterations have been implemented, the code is much less readable, and therefore more difficult to modify and debug. It seems likely that when comparing solution times between the original and modified versions using an unoptimised compiler, one would find the current version to be processed faster. This is indeed the case. However, in practice code should be compiled with full optimisation, to allow the compiler to make as many optimisations itself as it can. For problems of various size, the solution time per non-linear iteration (T_1) of the original code may be compared with the solution time per non-linear iteration of the now optimised code (T_2).

Figure 7 shows that the ratio of times between the streamlined code using the above methods, and the optimal readable code, when compiled using full optimisation, over problems of varying numbers of nodes.

For small problems, a noticeable reduction in computing time may be noted. For example, a mesh with 512 nodes shows a 23% reduction in execution time as a result of

the optimisations carried out. However, as the problem size increases, the savings in computing time become less marked. Thus for problems with many degrees of freedom, an alternative approach is necessary.

3.4 Mass Lumping

In the corrector stage of the solution process, much time is taken up by the solution of linear matrices. Zienkiewicz and Taylor [22] suggest that in implicit time-dependent FE formulations, the mass matrices are diagonalised so that no linear system of equations needs to be solved. This technique is known as mass lumping.

This part of the solution process projects the velocities onto a divergence free field. Therefore a modification such as this suggests that continuity will not be exactly satisfied at the end of each non-linear iteration, and this may have an implication on the accuracy of the solutions obtained. Case 2 is used to analyse this, running both the original algorithm and the mass lumped version. The Reynolds number used for this flow is 1.0, with relaxation values on the velocity and pressure of 0.6 and 0.2 respectively, from a start condition of $u=1.0$ everywhere so continuity is initially satisfied. The solution procedure was fully implicit, and 80 time steps were run, with 5 non-linear iterations per time step. The time step was 0.01.

At the end of the solution procedure, the pressure correction had dropped by four orders of magnitude, on all of the meshes though the primitives for mesh E had not converged. Calculating the mean error of the outlet velocity and centreline pressures from the analytical solution gives the discretisation overall error of the solutions and these are shown in Tables 5 and 6, for the original and mass lumped algorithms respectively. Note that values are given as percentage errors. The effect of mass lumping on the solution times across varying degrees of freedom is shown in Figure 8, using the Case 1 set of problems.

In terms of the accuracy of the solutions, compared to the analytical solution, it can be seen that increasing the number of elements in the streamwise direction has only marginal

effect on the accuracy of the solutions. This can be seen in the results for meshes A to C. However, increasing the elements in the spanwise direction does improve the solution. This can be seen by comparing the solutions of meshes B and D, and C and E in both tables. The unconverged results of mesh E typify a phenomena which has been observed repeatedly on problems: the finer the mesh, the more iterations are required to obtain convergence.

Comparing across Tables 5 and 6, the error induced by mass lumping is greater for coarse meshes. This is true for both the velocity and pressure, and has been verified for driven cavity flows. Therefore, for this flow, it appears that mass lumping has only a marginal effect on the overall accuracy of the solution, on fine meshes.

The general trend for the execution time per non-linear iteration would not be expected to change, as we are substituting the solution of seven equation systems per non-linear iteration for four. Similar trends are evident in Figure 8. Since the solution of linear systems dominates the execution time for increasing numbers of degrees of freedom, one would expect the ratio of execution times between the original algorithm and the mass lumped algorithm to approach 1.75. In fact, the asymptotic value appears to be somewhat less, at 1.43.

Mass lumping provides a reasonable way of maintaining accuracy and reducing the number of equation systems to be solved. However, in order to address the dominating time taken by this procedure, an alternative to the LU-decomposition routine needs to be found.

3.5 Iterative Solvers

Direct solvers solve linear systems such as equation (60), in a series of finite steps.

$$Ax = b \tag{60}$$

The number of steps involved is known in advance and does not depend on the matrix involved. The solution to the linear system is determined when the procedure is complete: early termination of the procedure is usually pointless. Moreover, for sparse matrices, direct methods may result in fill-in, whereby the structure of the sparsity is not preserved in intermediate calculations. This means that if a storage scheme is used then solutions may be affected. This does not appear to be the case with the LU-decomposition solver used.

Iterative solvers, on the other hand, start with an approximation to the solution of the linear system and then refine this solution according to some procedure, terminating when some criteria has been met. Many iterative methods do not require the matrix itself, except in terms of operations on other entities. This implies that any storage system can be as efficient as possible without having to make considerations for fill-in. Classical iterative methods are based on splitting techniques, and details can be found in Young [54].

Three classes of iterative techniques exist. These are

1. line relaxation methods,
2. symmetric conjugate gradient (CG) methods, and their asymmetric derivatives,
3. Minimum residual methods.

The line relaxation methods consist of the Jacobi, Gauss-Seidel (GS) and successive over-relaxation (SOR) methods. These are the simplest to implement, and require little storage. The Jacobi method, given by equation (61), only requires the solution from the last iteration, and in fact this is the most storage required by any of the line relaxation methods. This method is however, very slow to converge (see Young [54]). The GS method, given by equation (62), improves on this by immediately using values that

have been calculated in the evaluation of an unknown. This means that no previous solution needs to be stored, resulting in a simultaneous reduction of storage and improved convergence.

$$x_i^{(k+1)} = (b_i - \sum_{j=1}^n a_{ij}x_j^{(k)})/a_{ii} \quad (61)$$

$$x_i^{(k+1)} = (b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^n a_{ij}x_j^{(k)})/a_{ii} \quad (62)$$

$$x_i^{(k+1)} = \frac{\omega}{a_{ii}} \left[b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^n a_{ij}x_j^{(k)} \right] + (1 - \omega)x_i^{(n)} \quad (63)$$

where ω is the over-relaxation parameter, and $0 < \omega < 2$.

The SOR method, given by equation (63) improves further on the convergence of GS by over-relaxing the estimates of the solution. This has the same storage requirement as GS, but introduces an over-relaxation parameter, ω . The optimal value of ω may be evaluated from the spectral radius of the matrix, but this is expensive to calculate. However, since it is found that the speed of convergence of the SOR method is sensitive to this parameter, Chebyshev acceleration as recommended in Press *et al* [21].

CG methods are direct methods if exact arithmetic could be used, but are classed as iterative methods since reasonable solutions can be obtained in far fewer steps than the theoretical maximum number (see Howard *et al* [55]). The following approach is used. A function $\Theta(x)$ defined by,

$$\Theta(x) = \frac{1}{2}x^T Ax - x^T b \quad (64)$$

has a minimum value of $-\frac{1}{2}b^T A^{-1}b$, which corresponds to setting $x = A^{-1}b$. Thus minimising $\Theta(x)$ and solving equation (60) are equivalent problems.

The classic CG method is applicable only for cases when the matrix is symmetric and positive definite. For our solution scheme it has been found that the symmetry restriction still allows accurate solutions to be generated when the Reynolds number is low. This is to be expected because only the convective terms generate asymmetric contributions to the global stiffness matrix, and at low Reynolds numbers, these are close to zero. However, for increased Reynolds number, an asymmetric iterative solver is required.

Derivatives of the CG method exist which allow solution for asymmetric problems. These are listed in Appendix 1. Of these, the conjugate gradient residual (CGR), biconjugate gradient method (BICG) and the conjugate gradient squared (CGS) methods have been considered. Howard *et al* [55] shows that CGR methods are generally unsuitable for systems of equations generated by FE methods, unless re-orthogonalisation against all previous solutions can be afforded. Since this requires the storage of previously calculated residuals, this has not been implemented. The CGS method converges faster than BICG if the system of equations is well conditioned, and this has also been noted by Howard *et al* [55]. However, the conditioning of the matrices decreases with increasing Reynolds numbers, and therefore, preconditioning becomes necessary. Preconditioners are discussed in Section 3.6.

The minimum residual methods include ORTHOMIN and the GMRES algorithm developed by Saad and Schultz [56]. Of these methods, GMRES is reported to have better convergent properties than ORTHOMIN, but GMRES, like CGR requires storage of previously calculated residuals (see Howard *et al* [55]). Moreover, it is reported that minimum residual methods can exhibit instabilities after the method appears to be converging (see Habashi *et al* [57]). Note that CGR is mathematically equivalent to GMRES and ORTHOMIN in exact arithmetic (see Howard *et al* [55]).

3.6 Preconditioners

Iterative methods may be accelerated by use of a preconditioner which improves the condition of the matrix by improving the distribution of eigenvalues, by replacing the equation (60) with the equivalent system,

$$C^{-1}Ax = C^{-1}b \quad (65)$$

The non-singular matrix C is called the preconditioner and should have the following properties:

1. C approximates A such that $C^{-1}A$ has its eigenvalues clustered near 1.
2. For a given vector d , we can solve $Cy = d$ in $O(N)$ operations
3. C has low storage requirements

Preconditioners for CG methods have been discussed by Chin *et al* [58] and Turner [59]. The simplest preconditioner is diagonal scaling, where $C = \text{diag}(A)$. More complex preconditioners require extra work and storage. The Neumann preconditioners use the property that if $A = I - G$, where I is the identity matrix, then $A^{-1} = I + G + G^2 + \dots$. Denoting the right hand side of this as $G(n)$ (where n is the maximal exponent in the expansion), then $G(n)$ has the same shape as A . This condition is true only for small a_{ij} , necessitating scaling of the equation system such that each a_{ij} is divided by the maximum value of a_{ii} .

Comparing $G(2)$ with $G(1)$ we have found that for the CG method, the quadratic Neumann preconditioner results in faster convergence than the linear Neumann preconditioner. However, this requires calculation of a matrix-matrix product every time a

linear equation is to be solved. The cost of extra iterations for the linear Neumann preconditioner is less than the cost of evaluating this product, therefore suggesting the use of linear Neumann preconditioners.

Other preconditioners include incomplete Cholesky preconditioning, LU-decomposition preconditioning and SSOR preconditioning, although these have not been explored.

Some choices of C are based on the idea of incomplete Gaussian elimination in which a sparsity pattern is selected, and a normal LU-decomposition is performed. Any entries outside of the pattern are neglected.

3.7 Application of Iterative Solvers

The segregated nature of the formulation allows the use of different iterative solvers for each of the primitive variables. This allows exploitation of the characteristics of the global stiffness matrices produced. Specifically, the matrices generated for pressure are symmetric whereas those generated for the velocities are asymmetric. This has led to a two part iterative solution strategy for the matrix equations. The savings on the overall solution time using this type of strategy have been demonstrated (see Haroutunian *et al* [42], Haroutunian *et al* [35], and Mallick and Shaw [60]).

Initial implementations used a combination of SOR and CG. However, this was found to be inefficient compared to a combination of CGS and CG. For each of these systems of equations, preconditioning was necessary to facilitate solution.

3.8 Properties of Matrices

The effectiveness of the linear equation solvers is dependent on the properties of the matrix equations that it is trying to solve. It is well known that for matrices which are diagonally dominant and symmetric, solvers can be highly efficient, but as these

properties are removed from the matrix, the solvers become less reliable. Here we define various parameters for the diagonal dominance of the matrix and examine the ratios of the maximal and minimal eigenvalues for the velocity and pressure matrices generated at the first time step of various driven cavity problems.

Using the 'spy' function in MATLAB it is possible to examine the sparse structure of matrices. The global stiffness matrices generated for the two-dimensional driven cavity problem have a distinctive shape as shown in Figure 9. In this figure, an 8x8 element mesh has been used to discretise the domain. Whilst this would not give an adequately resolved flow solution as the mesh is far too coarse to generate a solution, the figure shows one consistent trait: if the domain is discretised into $n \times n$ elements, then the number of non-zero blocks in the upper and lower triangles is $n - 2$. Along the leading diagonal, there are $n - 1$ non-zero blocks of data, whilst the leading diagonal itself will always be fully populated. This is to be expected since the FE formulation is such that a non-zero value in the global stiffness matrix in position (i, j) represents the influence of node i on node j . Therefore the leading diagonal can have no zero values, and moreover, the global stiffness matrix is always shape symmetric.

The diagonal dominance of a matrix is defined in equation (66).

$$d_r = \frac{1}{n^2} \sum_{i=1}^n \left(a_{ii} - \sum_{i \neq j} \|a_{ij}\| \right) \quad (66)$$

For various sized driven cavity problems, d_r may be measured, shown in Figures 10 and 11. Clearly the diagonal dominance of the pressure matrices is much lower than the diagonal dominance of the velocity matrices. This suggests that iterative solvers will require more iterations to produce solutions for the matrices generated by the pressure equation. This is confirmed by comparing the ratio of eigenvalues of the velocity and pressure generated matrices, as shown in Figures 12 and 13. Ideally, the matrices should have eigenvalues clustered near 1, but in these cases, the ratio of eigenvalues of the pressure generated matrices is initially an order greater than those of the velocity generated matrices and

rapidly increases as the number of nodes in the problem grows. Note the log scale used in Figure 13. This further suggests that the number of iterations required for solving the pressure generated linear equations will be increasingly greater than those required for the velocity generated matrices as the problem size grows.

The actual matrix itself can be visualised using the meshing facilities of MATLAB. Figure 14 and 15 show the full matrices for the u-component of velocity and pressure respectively, for a driven cavity problem with 72 nodes. Again, it is clear that the pressure generated matrices are much less diagonally dominant than the velocity generated matrices.

3.9 Storage Schemes and Memory Access

The way in which data is stored, and retrieved during high computation subsections will affect the overall speed of the program. FE algorithms are such that the matrices which are produced tend to be large, but sparse. Therefore using a two dimensional array and storing all the data within it will be a very inefficient storage system, because of the unnecessary zeroes that will be contained. The matrices produced by the FE method are banded around the leading diagonal, and the bandwidth of the matrix is given by the maximum separation of non-zero elements in the rows of the matrix. Since the bandwidth of a problem may be only a fraction of the size of the matrix, storage schemes are necessary.

One storage scheme is that of skyline storage. In this system (used in the original program) the data is stored in skylines running down the leading diagonal (see Reddy [23] or Griffiths and Smith [51]). This produces an efficient way of storing the data but can lead to unreasonably long expressions to extract data. This was especially noticeable when running iterative solvers.

Two alternative storage systems are considered: diagonal and row storage. In the former, diagonals of the matrix are stored, indexed in one array. This has the disadvantage of storing the whole bandwidth of the matrix. An alternative is to store all the data in one

large array with rows of data packed sequentially. By assuming that each row of data has potentially differing numbers of entries, the overall storage requirement is reduced. This necessitates pointer arrays which may be pre-processed, thus requiring more book-keeping, but keeps memory requirements lower than would be possible with diagonal storage.

3.10 Discussion

For small problems, reduction of operation costs can be useful. Whilst the effect of this is reduced for implicit FE codes as problem size increases, because of the domination of solutions of linear equation systems, even on problems with over 2000 degrees of freedom, a 23% reduction of computing time is observed. This suggests that for explicit codes, savings would be much greater and may carry through even as problem size increases.

For implicit codes, mass lumping allows a convenient way of reducing the number of linear equation systems that are to be solved. On coarse grids, the accuracy may be compromised, but for finer grids, there is a negligible error.

For a matrix of order n , methods such as Gaussian elimination require $O(n^3)$ operations (see Press *et al* [21]). Moreover, storage requirements (for a structured mesh) are proportional to $O(n^{1.67})$ (see Habashi *et al* [57]). Iterative methods offer storage requirements of $O(n)$, and for well conditioned matrices CG methods for example, can produce machine accurate solutions in $O(n^{1.17})$ operations.

Thus for large problems, using mass lumping and iterative solvers rather than direct solvers is appropriate.

n	Number of nodes	Number of elements	Number of boundary conditions	Size of skyline matrix
2	27	8	79	235
4	125	64	295	3101
6	343	216	655	16759
8	729	512	1159	58969
10	1331	1000	1807	160931
12	2197	1728	2599	371125

Table 2: 3D Driven cavity mesh statistics. These meshes are used to determine timing characteristics of a solution procedure over problems of varying degrees of freedom, representing the Case 1 set of problems for Chapter 3. This table is referred to in Section 3.1.

Mesh label	Element layout	Number of nodes	Number of elements
A	10x8	99	80
B	20x8	189	160
C	40x8	369	320
D	20x16	357	320
E	40x16	697	640

Table 3: Pipe flow mesh statistics. These meshes represent the Case 2 set of problems for Chapter 3 and are used to compare accuracy with different solution methods. This table is referred to in Section 3.1.

	Times on:		
Operation	SunSparc Classic	SunSparc Station 10	2 linked Sparc Station 10s
Empty do loops	4.4	4.6	3.6
Replacement	5.1	4.9	3.9
Scalar operations			
+ and replace (FL)	6.8	5.6	4.6
- and replace (FL)	6.8	5.6	4.5
and replace (FL)	7.0	5.6	4.5
/ and replace (FL)	10.0	6.5	5.2
Integer IF	5.4	6.0	4.7
FL IF	7.6	7.0	5.9
Array operations			
+ and replace (FL)	9.3	7.9	6.0
- and replace (FL)	9.3	7.9	5.8
and replace (FL)	9.5	8.4	5.9
/ and replace (FL)	12.4	8.7	6.1
FL IF	8.8	7.9	6.1

Table 4: Operation timings on various machines. In this table FL refers to floating point operations. Values represent times in seconds required to perform 10 million operations in single precision when machines are run in single user mode. Machine dedication for all times is 99% or greater. See Section 3.3.1.

Mesh	Velocity error	Pressure error
A	3.53	4.26
B	3.36	4.23
C	3.34	3.99
D	1.91	2.09
E	1.97	2.13

Table 5: Monitor values for pipe flow experiments with no mass lumping. This table is referred to in Section 3.4.

Mesh	Velocity error	Pressure error
A	3.55	4.72
B	3.35	4.14
C	3.33	4.04
D	1.91	2.10
E	1.97	2.14

Table 6: Monitor values for pipe flow experiments with mass lumping. See Section 3.4 for a discussion of the values in this table.

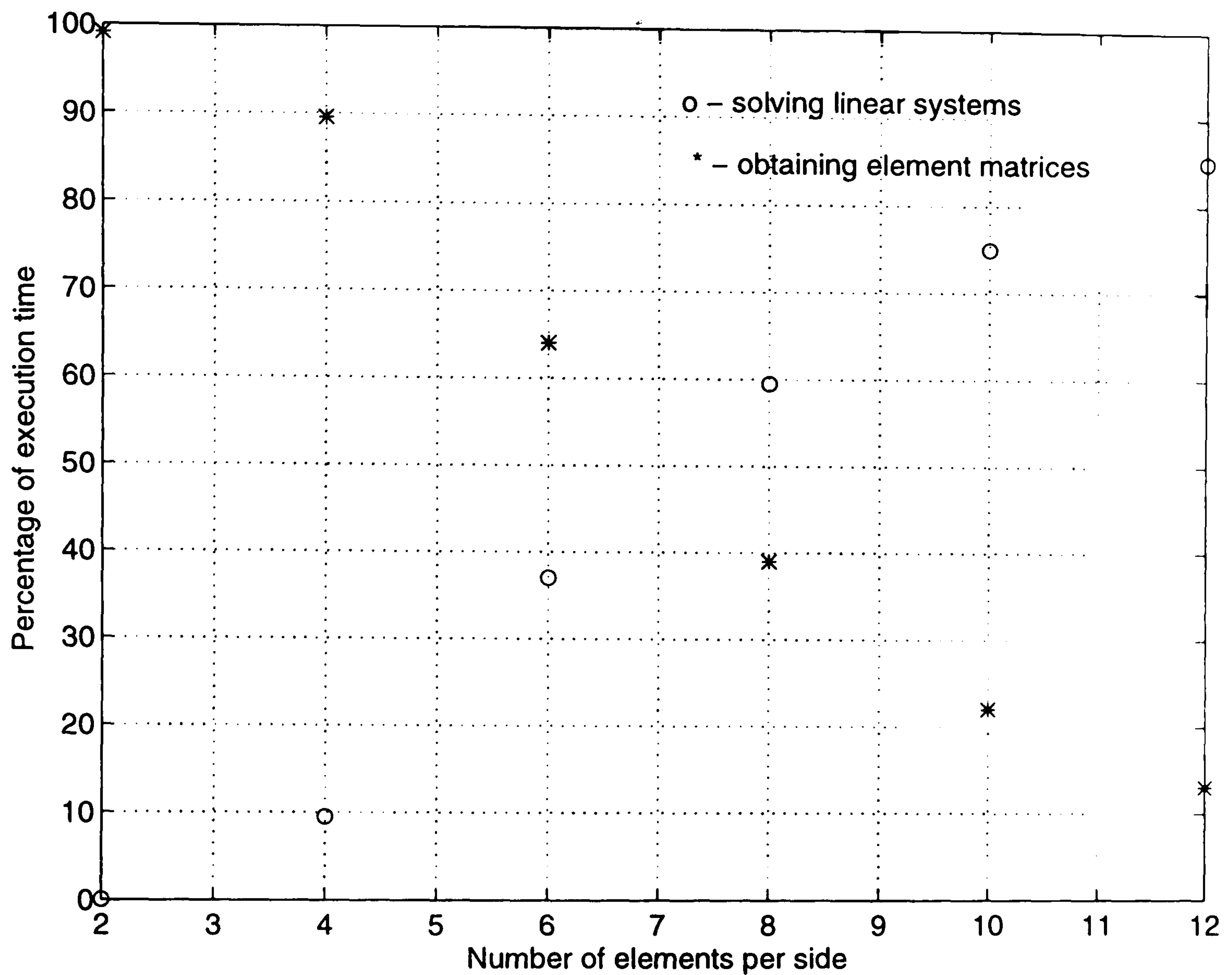


Figure 6: Domination of solution time by linear equation solvers as problem size increases. For three dimensional driven cavity problems, with an increasing number of elements per side, the time taken to solve the linear systems of equations takes an increasing percentage of the total execution time. This is further discussed in Section 3.2.

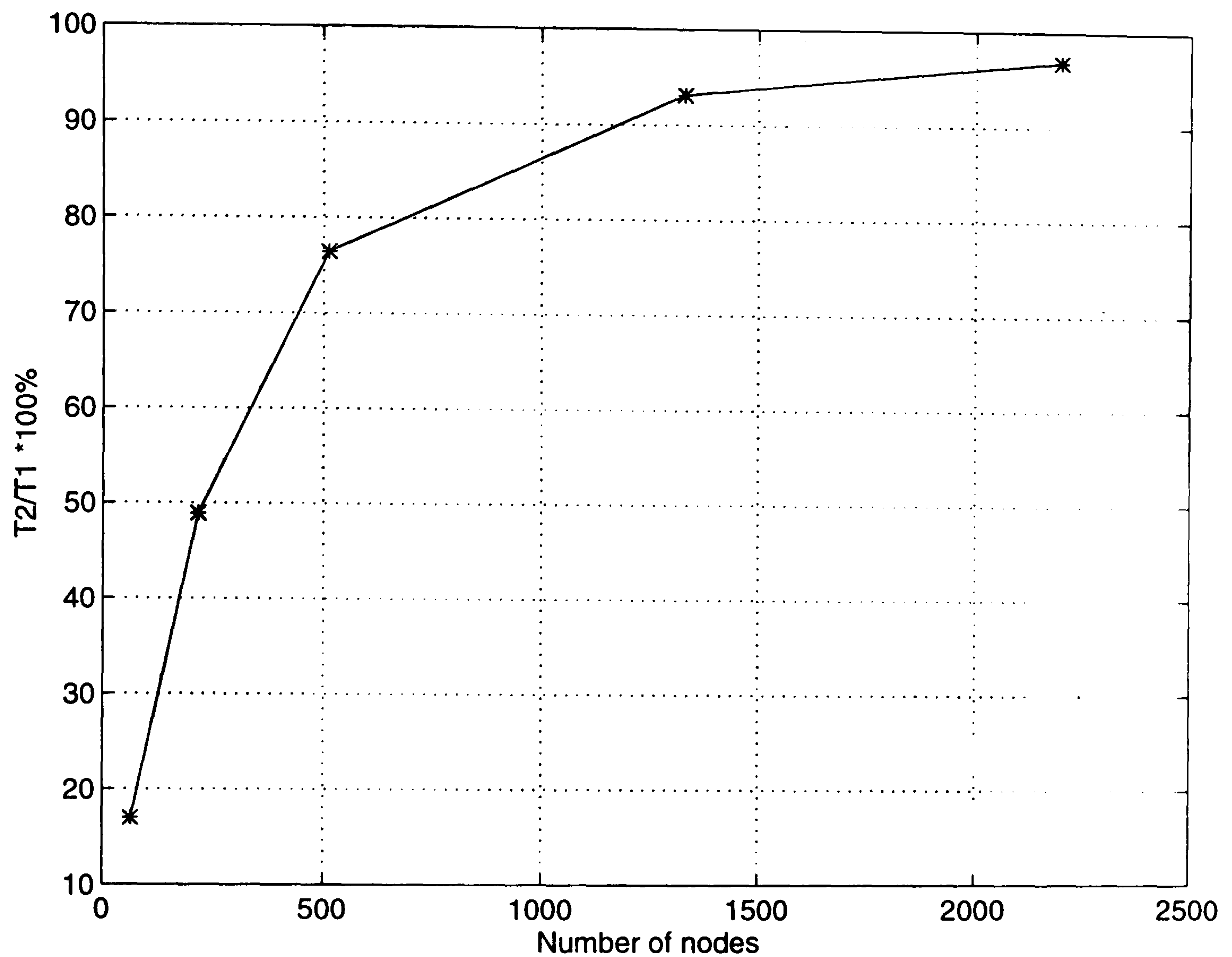


Figure 7: Effect of some computational reductions. This figure shows the ratio of the solution time per non-linear iteration (T_2) of a streamlined code and the solution time per non-linear iteration (T_1) of the original code, against increasing problem size. See Section 3.3.2.

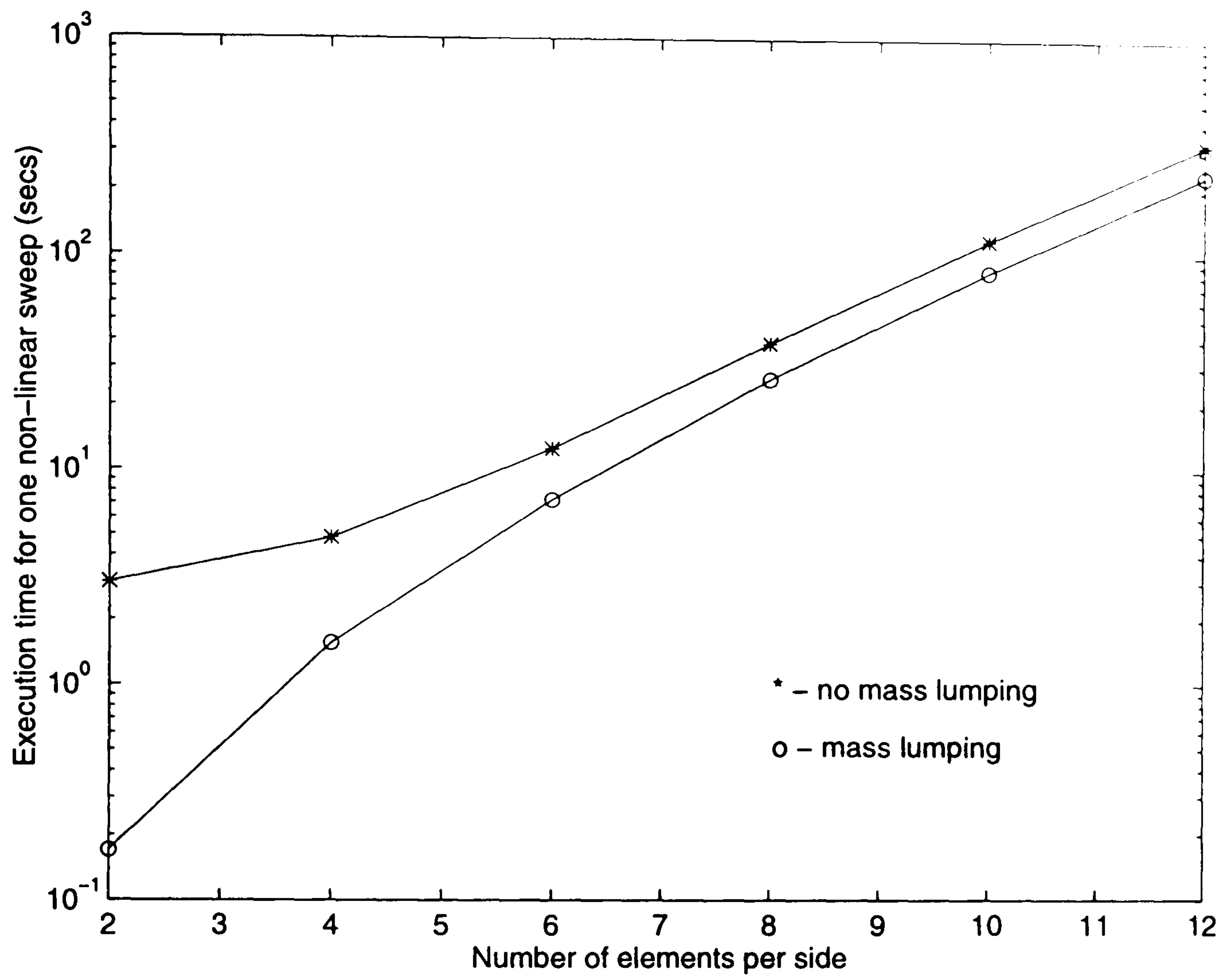


Figure 8: Comparison of execution times of solvers running with and without mass lumping. The execution times for one non-linear iteration are shown over problem size. See Section 3.4.

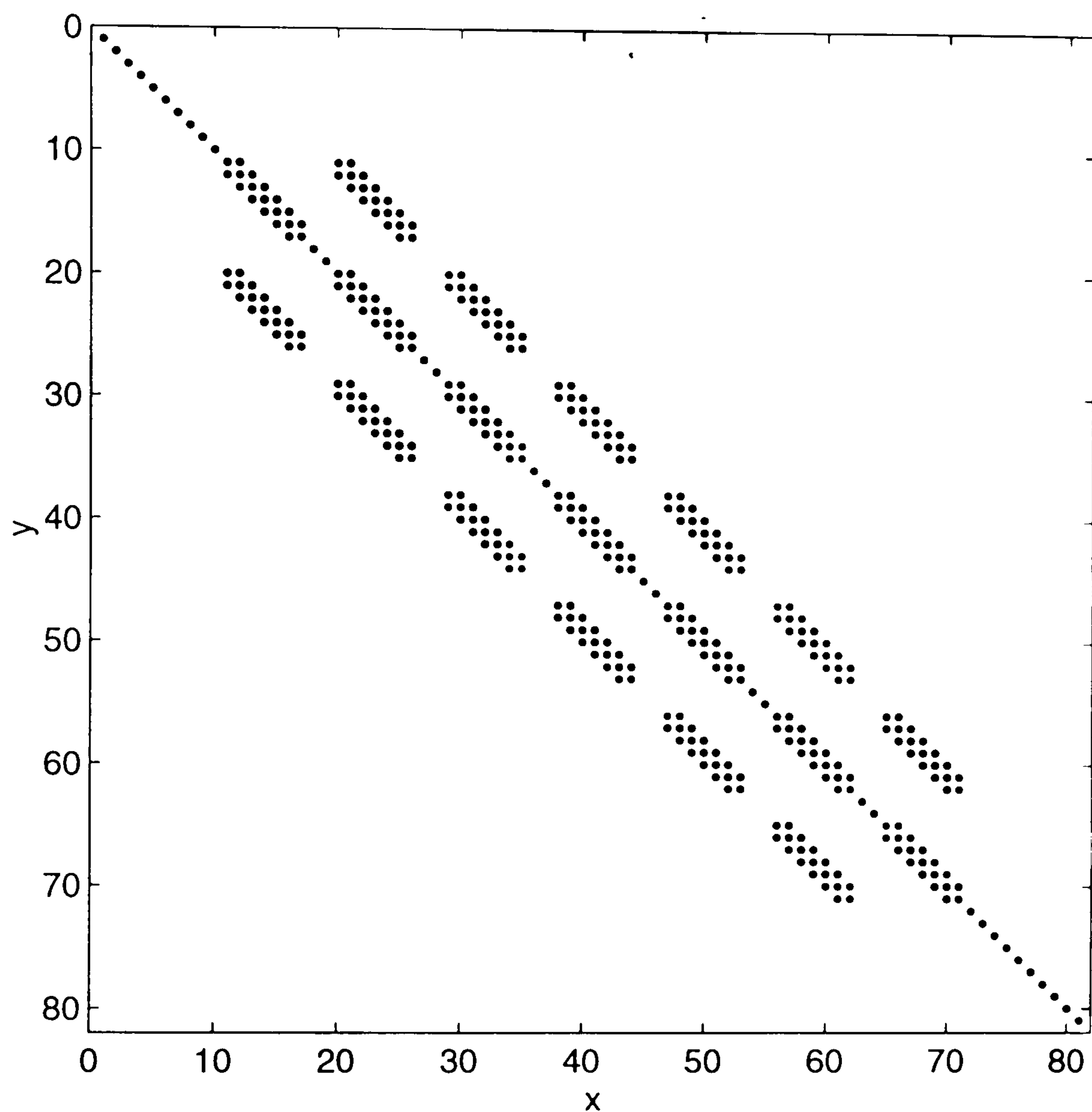


Figure 9: An example of the sparsity pattern for 2D driven cavity problem. The figure shows the non-zero elements of the global stiffness matrix, for an 8×8 element mesh. This gives the mesh 81 nodes, hence the dimensions of the matrix. This figure is discussed in Section 3.8.

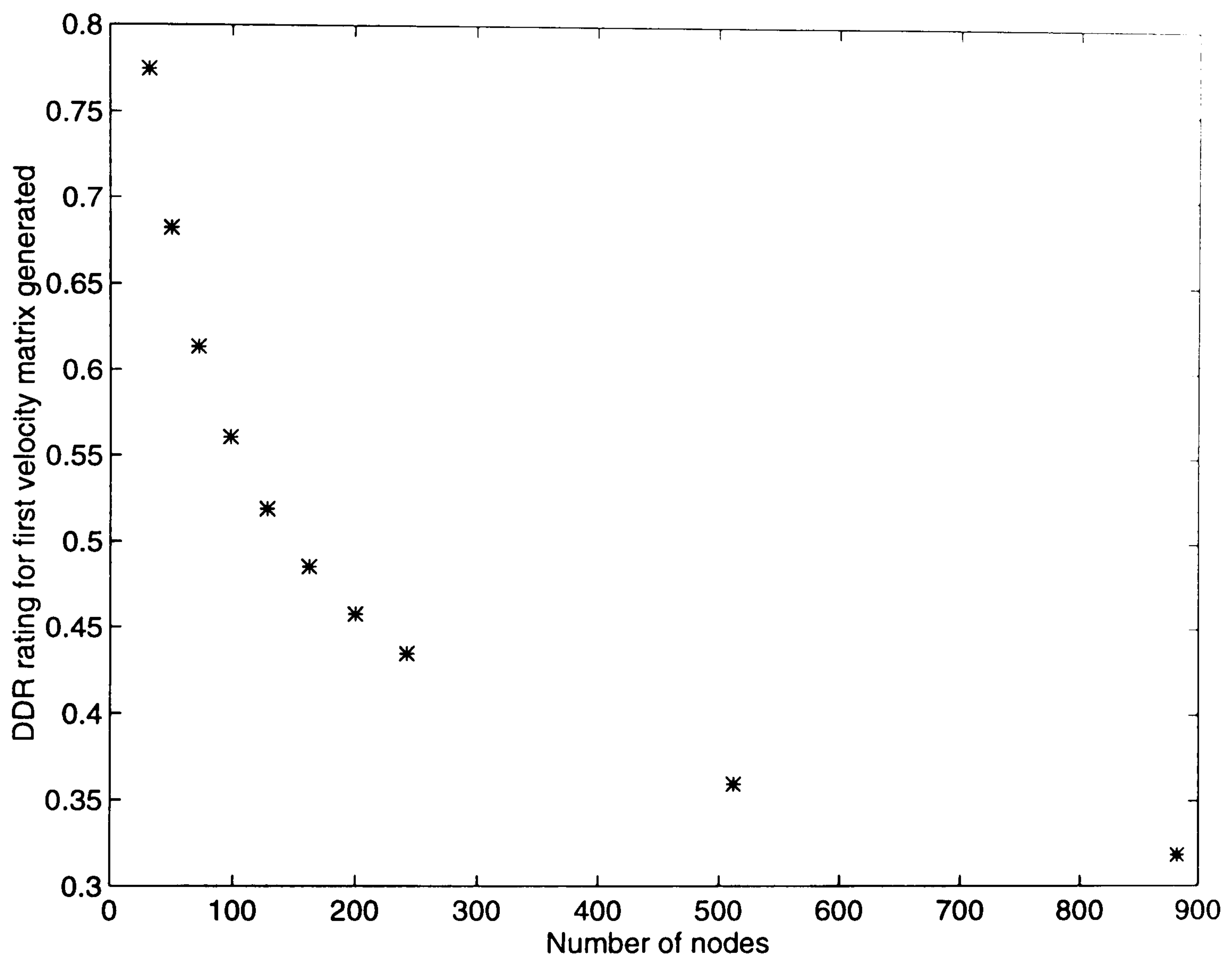


Figure 10: Diagonal dominance ratings (DDR) for velocity generated matrix. Using the definition of DDR defined in Section 3.8, this figure shows the reduction of diagonal dominance for increasing problem size of the first matrix to be generated from the u-momentum equation, for various driven cavity problems.

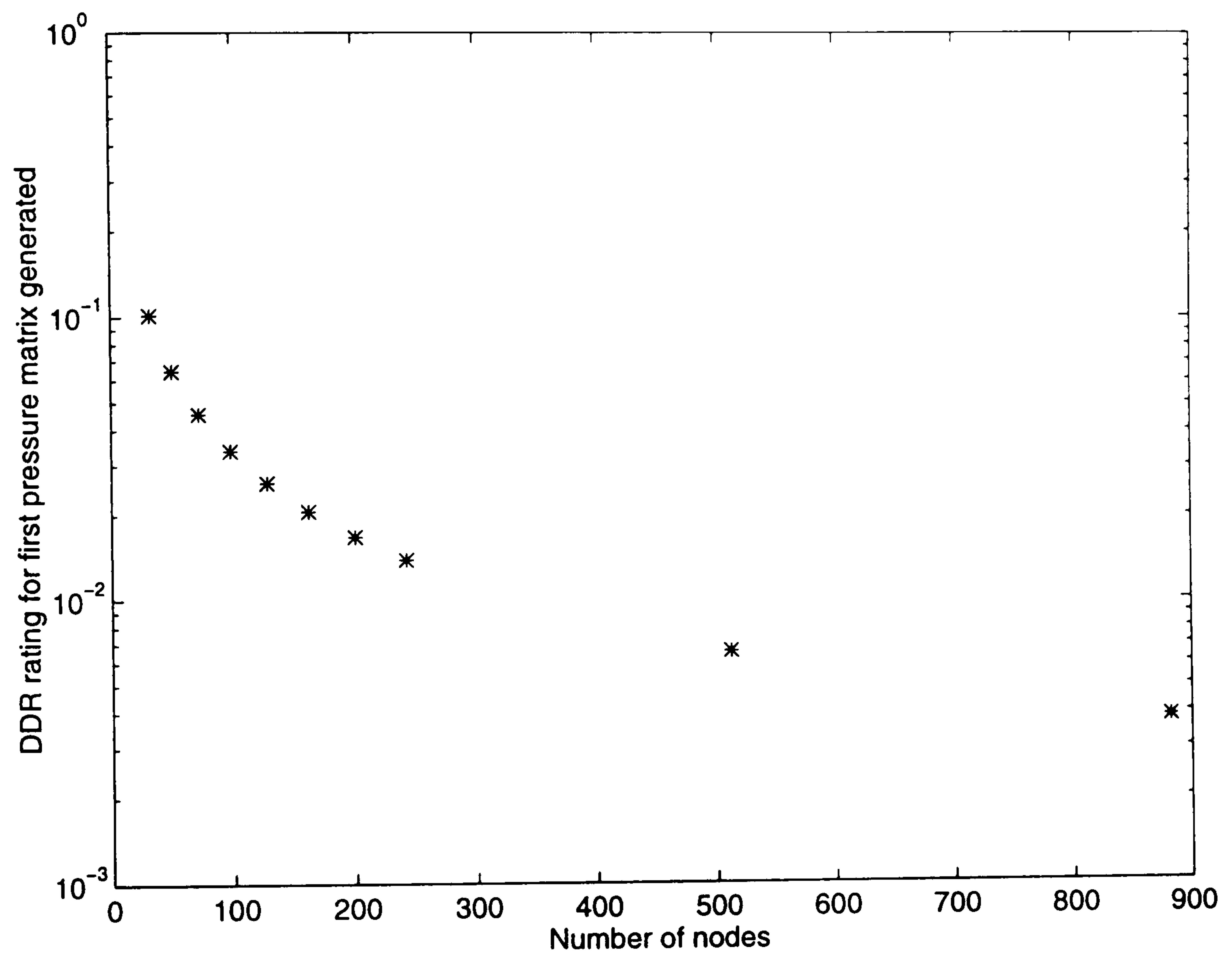


Figure 11: Diagonal dominance ratings for a pressure generated matrix. This is discussed in Section 3.8.

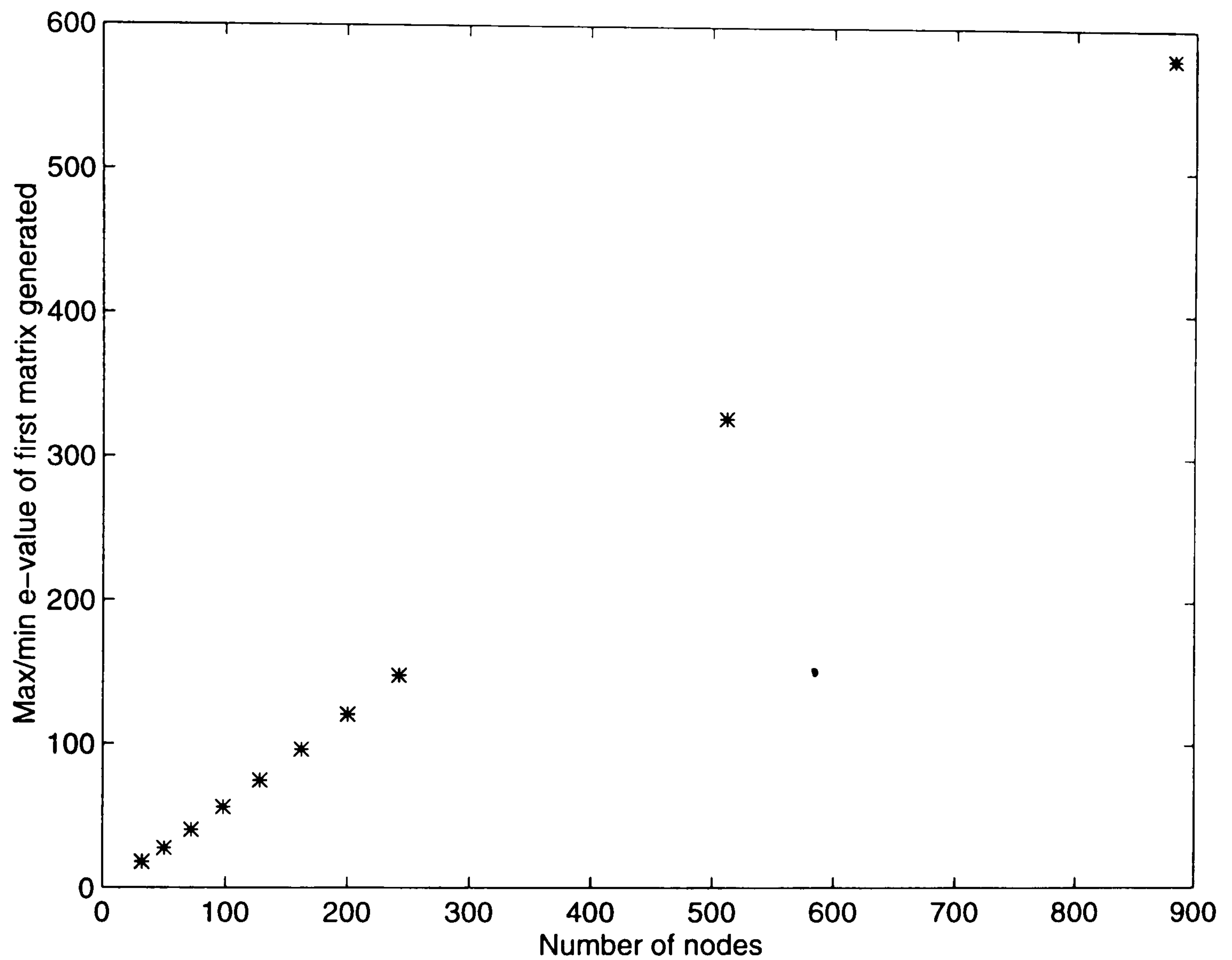


Figure 12: Ratio of eigenvalues for velocity generated matrix. This figure shows the ratio of the maximum and minimum eigenvalues for the first matrix to be generated from the momentum equations for various driven cavity problems. See Section 3.8.

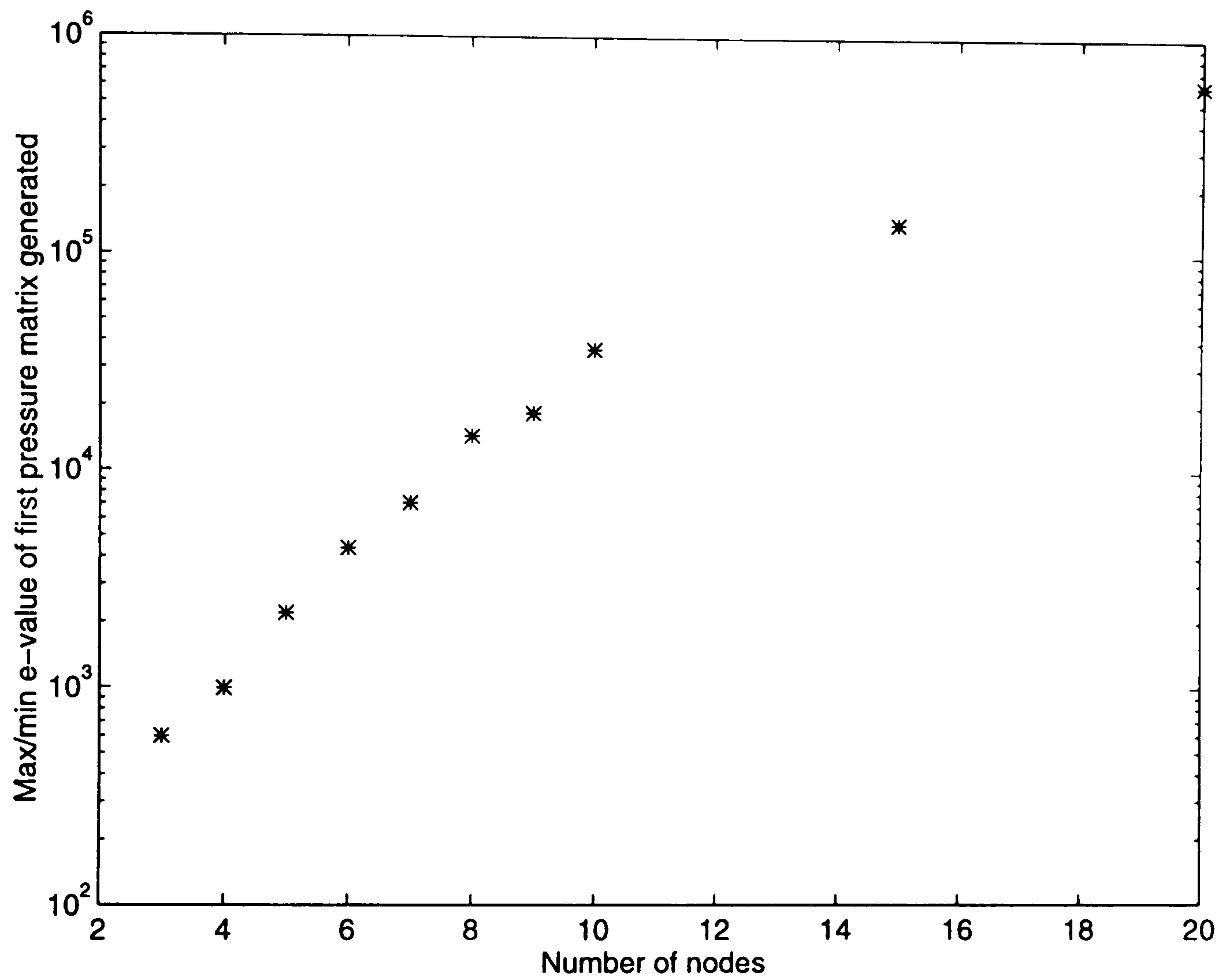


Figure 13: Ratio of eigenvalues for pressure generated matrix. This figure shows the ratio of the maximum and minimum eigenvalues for the first matrix to be generated from the pressure correction equation, for various driven cavity problems. See Section 3.8.

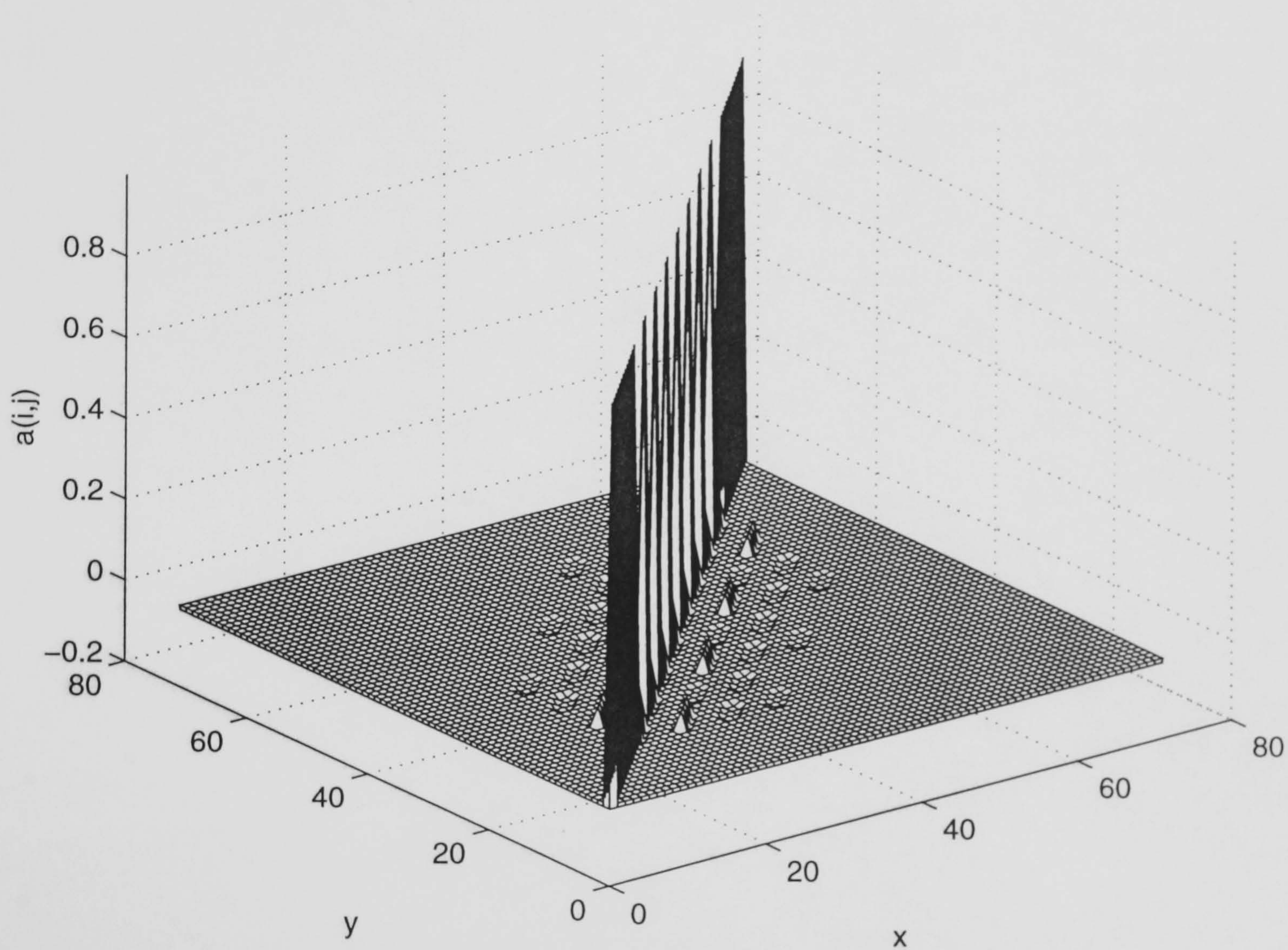


Figure 14: Diagonal dominance of velocity generated matrices. This figure is a visualisation of a full matrix generated from the u-momentum equation for a driven cavity problem with 72 nodes. It is clear that this matrix is diagonally dominant, as discussed in Section 3.8.

4 PARALLELISATION

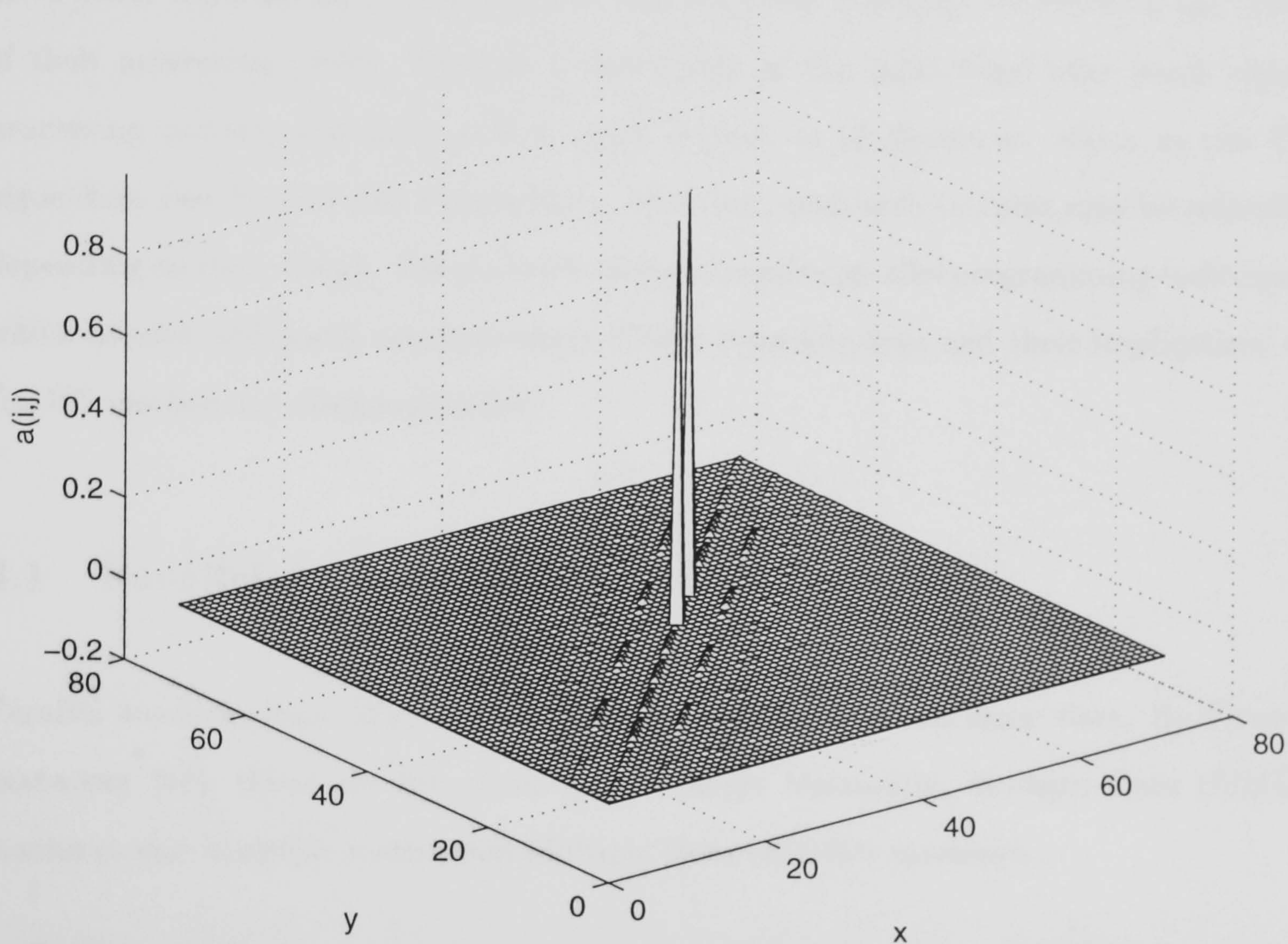


Figure 15: Diagonal dominance of pressure generated matrices. This visualisation of the matrix generated from the pressure equation shows the relatively small diagonal dominance when compared with that of the velocity. The matrix is from the first pressure generated matrix in a driven cavity problem with 72 nodes, as discussed in Section 3.8.

4 PARALLELISATION CONSIDERATIONS

Despite the optimisation of the sequential code, the actual solution time required for problems with many degrees of freedom is prohibitively long. This is a problem common for many academic and industrial applications. Sequential architectures are not able to meet many computational requirements, and yet these machines are reaching the limits of their processing power. Parallel architectures on the other hand offer much higher processing powers, and have gained much respect as platforms on which to run FE algorithms (see Fischer and Patera [61]). Moreover, such architectures may be scaleable depending on their design. Parallel architectures require parallel programming techniques which involve additional considerations. These considerations and their implications to the FE method are discussed below.

4.1 Parallel Architectures

Parallel machines have more than one processor working at the same time. By Flynn's taxonomy [18], there are two main types: Single Instruction Multiple Data (SIMD) machines and Multiple Instruction Multiple Data (MIMD) machines.

In SIMD machines a program runs on a single machine (the front-end) which broadcasts the same instruction to multiple processors, in any one clock cycle. These in turn apply the instruction to their own data set. Thus all processing occurs in one control thread, in a synchronised manner, which is known as lockstepping.

Though the processors operate in lockstep, each processor has some degree of autonomy, in that any number of the processors may be masked out, that is performing no operation, whilst the remaining processors are activated. This important feature allows selective processing of conditional statements based on some criteria. One example of this is an IF-THEN-ELSE construction. If the conditional is satisfied on only a proportion of the data, then only the corresponding processors are activated to perform the instruction.

Then, as the front-end processes the ELSE action, the remaining processors become active. Typically, SIMD machines produce a high computational speed by having large numbers, say 1024 or more, of slow (and cheap) processors.

MIMD architectures, are asynchronous, in that sub-problems are distributed over autonomous processors allowing multiple threads of control. These processes, or sub-problems, must exchange information, thus requiring synchronisation unless the processes are totally independent. Synchronisation induces costs, and should therefore be kept to a minimum. MIMD architectures typically have a relatively small number of fast (and expensive) processors.

MIMD architectures may be further categorised into shared memory (also known as tightly coupled or multiprocessor) machines and distributed memory (or loosely coupled) machines (see Braunl [62]). Parallelisation techniques for each of these differ because of the manner in which communication occurs. In shared memory architectures, the processors are linked only by virtue of using the same memory, via a bus. Thus memory contention may be a major problem. Distributed memory machines differ in that communication occurs over a network; thus synchronisation and communication costs are higher (see Braunl [62]).

Two types of parallelism have been defined (Hockney and Jesshope [63]): control (or process) parallelism and data (or structure) parallelism. In control parallelism, problems are divided into separate tasks that may be run concurrently. This type of parallelisation can only be used on MIMD machines. With data parallelism, concurrency is achieved by performing the same instruction on different items of data distributed across processors. This strategy, typically used on SIMD machines has recently been applied to MIMD architectures, resulting in a Same Program Multiple Data (SPMD) strategy (see Fischer and Patera [61]). This allows SIMD programming style to be combined with the MIMD flexibility of multiple control threads. Each processor of the system executes the same SIMD program on its local data with an individual control flow, allowing switching to occur between lockstepping SIMD type control, and asynchronous MIMD control. This

means that some of the limitations that the SIMD model has, for example, with IF-THEN-ELSE constructions do not result in a loss of efficiency, as processing may occur on all processors for differing actions of conditionals. An application of this model may be found in Oden and Patra [64].

4.2 Performance Measurements

The performance of a parallelised implementation may be measured by its speedup and efficiency. However, differing definitions of these parameters are commonly used. We define the parallelisability of a program on p processors, $P(p)$, the speedup on p processors, $S(p)$, and the efficiency $\eta(p)$ below:

$$P(p) = \frac{T(\text{parallel program on one processor})}{T(\text{parallel program on } p \text{ processors})} \quad (67)$$

$$S(p) = \frac{T(\text{sequential program on one processor})}{T(\text{equivalent parallel program on } p \text{ processors})} \quad (68)$$

$$\eta(p) = \frac{S(p)}{p} \quad (69)$$

where $T(.)$ is the time taken for a program to run.

The parallelisability of an algorithm gives a measure of how well an implementation responds to increases in the number of processors. Some authors have referred to this as the speedup, however, this may be misleading because overheads which have been introduced to allow parallel processing are masked (see Braunl [62]). The speedup as defined above takes into account the parallel overheads, and allows direct comparison with the parallel implementation and an optimal sequential algorithm. Note that the numerator of equation (68) may be difficult to obtain directly, due to memory limitations

of one processor in a parallel machine. If $E(\text{machine A})$ is the rate of execution in MFlops on machine A, then the numerator of equation (68) may be indirectly derived as:

$$T(\text{program on machine B}) = T(\text{program on machine A}) * \frac{E(\text{program on machine A})}{E(\text{program on machine B})} \quad (70)$$

Our definition of $S(p)$ is sometimes referred to as scaled speedup (see Quinn [17]), however, obtaining a true experimental value is usually impossible for the above reasons.

4.3 Choice of Architecture

Algorithms have two degrees of parallelisation corresponding to a data parallel or control parallel methodology. Due to the lockstep manner of SIMD programming models, which may be viewed as a restriction of the MIMD model, the degree of parallelisation for a SIMD code is less than or equal to the degree of parallelisation for a MIMD code. This is offset by an increased number of processors in SIMD architectures. Thus efficiencies of SIMD applications are typically less than those of MIMD applications (see Braunl [62]), which suggests that MIMD architectures may be appropriate, if high efficiencies are crucial.

However, in the application of the FE method, three features of SIMD architectures make their use desirable. SIMD machines are easier to program, since problems of memory contention and synchronisation do not exist. This is because each processor has its own local memory and because of the lockstep nature of calculations. However, the lockstep restrictions mean that optimal performances are difficult to achieve. Also, if p processors execute (almost) identical sequence of instructions (that is, taking the same route through a program) then p identical instructions are fetched from memory into p identical instruction processing units (IPUs), when only a single instruction fetch, IPU, and broadcast is necessary. Thus both communication bandwidth and memory are being wasted. If an implementation can be constructed which exploits parallelism

across the structure of the data, then a SIMD machine is appropriate. In general, FE methods require a sequence of the same operations applied to large data sets, suggesting SIMD usage. For other methods, where the intrinsic structure of the data is difficult to exploit, a MIMD machine may be more appropriate. Furthermore, when one is concerned with solving as large problems as possible, scalability becomes an important issue. An algorithm is linearly scaleable if the time taken for execution increases linearly with the problem size, whereas an architecture is scaleable if the performance per processor remains constant as the number of processors increases. Data parallel algorithms are more scaleable than control parallel algorithms, since parallelisation is targeted across the data of the problem. Moreover, SIMD architectures scale better than MIMD architectures (see Brauni [62]). For these reasons we select a SIMD approach.

Nevertheless, MIMD implementations have been reported, and discussions on shared memory architectures may be found in Sawley [65], whilst discussions of distributed memory MIMD architectures may be found in Barragy and Van de Geijn [66] and Natarajan and Pattnaik [67]. Some MIMD strategies coincide with common SIMD strategies with respect to the FE method, and these will be discussed in Section 4.5.

4.4 Manchester's MasPar Machine

Our target SIMD machine is the MasPar (MP1104), shown in Figure 16. This consists of a front-end workstation (a standard DECstation 5000 Model 200) which acts as host to the Data Parallel Unit (DPU). The DPU is based on a two dimensional mesh topology arranged in a 64x64 grid of processor elements (PE), giving a total of 4096 processors. Each PE is a custom-made 1.8 MIPS RISC-like processor, connected to 16 Kbytes of local memory. This gives the DPU a theoretical processing speed of 145 MFlops (double precision) and a total of 64 Mbytes.

Three forms of communication exist. Besides data transfer between the front-end and the DPU (which is the slowest form), inter-processor communication (IPC) can occur

either via a global router (which consists of a three-stage switch) or via the X-net (which links every processor with its eight neighbours, wrapped around at the physical edge of the processor grid). The global router has a throughput of 1.3 Gbytes/s whereas X-net communication is capable of 23 Gbytes/s. Therefore the communication method used can affect the solution time by as much as an order of magnitude.

However, IPC is transparent to the programmer. The manner of communication is decided (using the high level language MPFORTRAN, which is based on FORTRAN 90) at compile time. This decision is based on how program statements are formulated, and on how the arrays in the program statements are mapped.

Array mapping can be explicitly defined to place a variable either on the front-end or on the DPU, using compiler directives. Referencing an array contrary to its mapping directive results in whole arrays being passed across a data bus which links the front-end to the DPU. This phenomenon (called *sloshing*) is to be avoided where possible, because processors are idle whilst communication is occurring. This precludes the possibility of using the front-end for processing array variables that have been mapped to the DPU unless no parallel alternative can be found. Thus, even sequential processes have to be parallelised (for example, the imposition of boundary conditions). Additional specifications may be found in Appendix B.

4.5 Parallel Approaches to Finite Element Algorithms

Parallelisation of a FE scheme is non-trivial. The mesh needs to be mapped to the processors, such that the workload on each of the processors is roughly equal (the *load balancing* problem). This mapping will affect not only storage costs, but also the communication requirements. Whilst an efficient storage system will reduce memory requirements, it may result in computationally demanding retrieval expressions, reducing overall efficiency (the *data storage* problem). Implicit schemes generate a system of linear equations that need to be solved at every iteration. The matrices involved are large and sparse and, and

may require storage. Moreover, the solution of the linear system is challenging because of the communication overheads induced by the arithmetic of both direct or iterative algorithms. One further difficulty is the imposition of boundary conditions.

4.5.1 Load balancing problems

For domains with regular structured grids, which are to be solved on SIMD architectures, the standard method of mapping technique is to take equal sized partitions of the mesh and map each partition to a processors. This mapping is done such that geometrically adjacent partitions are placed on physically adjacent processors (see Fischer and Patera [61]). In iterative procedures, three computational tasks are necessary. These are calculations involving a discrete operator acting on a known vector of nodal values, the weighted update of primitive variables based on known vectors, and the calculations of inner products between two vectors. The first of these may be calculated by a global matrix-free technique, involving the calculation of the local elemental matrix-vectors, followed by an exchange/summation procedure of the finite element nodal values on the edges of the processor subsquares, first east to west, and then north to south.

For domains which are globally unstructured, but comprise of simply connected, regular subdomains, a domain decomposition (or multiblock) approach may be used. The discretised equations are solved over each block resulting in a series of almost independent sub-problems, necessitating communication only for edge information. Some methods allow domains to overlap. However, load balancing is more difficult especially if partitioning into subdomains is done by geometrical considerations.

This allows the domain to be partitioned into structured subdomains which may be mapped to a subset of the processors. Thereafter, calculations may be carried out in much the same fashion as for regular structured grids, with communication exchanges necessitated at the edges of partitions. This approach is used for both control parallel methodologies, (see Sawley [65]) and data parallel implementations (see Sawley and

Tegner [68]).

For truly unstructured problems, domain decomposition methods have been extended so that a mesh is broken down into collections according to mapping techniques. Most of these aim to form collections of roughly the same size, but neglect communication costs, for example, simulated annealing. Another class of techniques aim further by forming collections of the same size with a minimum number of collection edges with the view of reducing communication. Techniques of this class include the greedy algorithm and recursive spectral bisection (RSB). These are reviewed in Fischer and Patera [61] and have been applied by Kennedy *et al* [69] on the CM-5, who found that RSB produced superior decompositions, though Fischer and Patera [61] note that significant processing time is required to generate the mesh to processor mapping.

4.5.2 Data storage

Meaningful data representations may be obtained by suitable choice of elementary objects, over which the parallelisation should occur (see Johnsson and Matthur [70]). An elementary object may range from, a subdomain of the mesh (as in domain decomposition methods), to a clustered group of elements (see Liou and Tezduyar [71]), to an individual unassembled element (see Hughes *et al* [72]). Johnsson and Matthur [70] go further by investigating two different kinds of elementary objects: an individual nodal point *and* an individual nodal point within an element. These selections define the data storage for many of the variables in an algorithm. For those remaining a decision must be made for the data structures which will determine communication costs and the efficiency of the storage system used, in relation to the choice of elementary object. Johnsson and Matthur [70] consider these costs in relation to the minimum cost of communication from processor to processor allowing explicit communication constructs to be defined, but this requires low level programming.

For the global stiffness matrix, an efficient storage scheme is essential if the matrix is

to be fully assembled. Early schemes performed this assembly (for example, Lai and Liddell [73]) typically using a diagonal storage scheme in which each diagonal of the matrix is stored on a separate processor. However, explicit assembly may be unnecessary, depending on the choice of linear equation solver. For iterative techniques, the action of the matrix on vectors needs to be evaluated. This can be obtained in various ways, though the most common is through choice of elementary objects which are element based. This allows calculation of elemental matrix-vector products which may be summed to produce the required global matrix-vector value. Again, the level of communication involved in this calculation is dependent on the storage schemes for the other variables.

Jacobsen [74] suggests in passing, that data can be remapped into a more effective form and then returned to the original positions (if required) upon completion. Using the concept of elementary objects, this is effectively allowing multiple objects to exist through the algorithm, and projecting objects from one level to another. However, no investigations of this possibility were made.

4.5.3 Solution of linear equations

Solutions of linear equations may be evaluated through either direct or iterative methods. For direct approaches of tri-diagonal systems, efficient parallel solvers have been developed which exhibit almost full speedup (see Lang [75]). However, for arbitrarily banded matrices, fill-in may occur during the solution process, which may necessitate an additional storage scheme of at least the same size as the assembled global stiffness matrix. Parallel implementations of direct methods are not considered, though a discussion of this topic may be found in Habashi *et al* [57].

Three families of methods are considered: the line relaxation methods (Jacobi, Gauss-Seidel, and SOR), the CG methods (PCG, CGS) and the minimisation of residual methods (GMRES). In all of these methods, the main difficulty in application is the efficient evaluation of the matrix-vector product. This calculation will be considered in detail in

the Chapter 5. Aside from this the following properties may be noted.

Of the line relaxation methods, the Jacobi is the simplest to parallelise but has very slow convergent properties. The SOR techniques are the best of this family of methods, but parallel implementation is not straightforward as each evaluation for an arbitrary element of the solution vector requires the solutions for previously calculated elements. The method therefore requires colouring to allow the algorithm to function in a data parallel way, as described by Quinn [17]. Red-black schemes are an example of this, in which alternate processors are considered to be red/black. Then, for example the estimates for elements in the solution vector corresponding to red processors are updated, after which the estimates corresponding to the black processors are evaluated. Inevitably this results in a processor utilisation of no more than fifty percent, but nevertheless this methodology allows implementation of the SOR techniques.

Implementation of the CG methods is straightforward (given an efficient matrix-vector product), but the classical methods are only applicable to symmetric matrices. For asymmetric matrices, extensions of the BICG method are necessary. Of these, the CGS method is a reasonable choice as it has reasonable storage requirements (6 vectors) and is fairly robust.

The GMRES method has been used frequently on sequential architectures; however, it has been shown that minimum residual methods are not robust for non-positive systems and can stagnate at non-zero values of the residual. Moreover, GMRES requires the storing of previously calculated solutions, and it is often necessary to store up to 20 of these in order to calculate the next search direction. Therefore GMRES is not considered further.

Of the iterative methods the CG methods appear to be the most suitable for parallelisation, given their low storage costs and relatively simple implementation, robustness and generality, coupled with reasonable rates of convergence. Moreover, because of the segregated nature of Shaw's algorithm (see Shaw [19], [20]), different solvers may be used to

exploit the symmetry of the linear systems produced by the pressure solver. This would not be possible for a traditional coupled scheme.

4.5.4 Imposition of boundary conditions

Imposing the boundary conditions is essentially a sequential process. In parallel, imposition is dependent on the data structures of the stiffness matrices, and whether or not it is explicitly assembled. Regardless this process will inevitably result in inefficiency, but should take only a small proportion of the time of the total calculation.

4.6 Discussion

Sophisticated methodologies for the domain decomposition have been developed and applied. However, whilst minimisation of the number of edges of the collections and reasonable load balance is desirable, efficient application of these decomposition methods necessitates explicit configuration of communication pathways. This may only be done using low level language constructs. However, such constructs require specialised knowledge of a particular architecture, and results in a machine specific code.

Ideally a FE implementation should be portable across platforms within a class of architecture. This implies that machine specific constructs must be avoided, and whilst this may result in implementations which are not as efficient on a particular machine, it allows for rapid programming of codes and facilitates the possibilities of building software libraries across platforms.

The restriction of using only high level constructs implies dependence upon so-called *intelligent* compilers, which decide at compilation time the mode of communication that occurs for a particular line of code. This decision is based upon the mapping used for variables with the statement and the specific arithmetic to be performed. This allows the benefit of general programming techniques which are able to benefit from improvements

in compiler technology. However, this in itself is not without potential pitfalls. For example, in one version of our code, a subroutine contained a statement of the following type:

```
xtemper=xsol(jpos(ihs:ihe))*dinfo(ihs:ihe)
```

This is not an unusual statement; vector addressing is used within an expression necessitating some interprocessor communication. Running on MasPar using the MPFORTRAN compiler version 3.2 resulted in this particular statement taking 20 times longer than it had done for compiler version 3.1, using the same code and input data. Detection of this problem was difficult because of bugs in a prototype profiling routine (version 3.2 Beta) which incorrectly reported timing counts for some statements. Nevertheless, once the problem was identified another version of the compiler was provided, and the official profiling software release rectified profiling problems.

Experience with the MasPar has shown that the difference between an implementation and an efficient implementation is largely due to the communication costs involved in processing instructions. Whilst some work has been done in minimising communication in a formal manner, these methods have required low level programming.

Very little work appears to have been done on the minimisation of communication via solely high level programming. Such methods would allow relatively simple code production and would be applicable to parallel machines within the SIMD class.

Johnsson's concept of the elementary object allows an intuitive representation of the data that exists on a processor. However, it is not clear how the selection of a particular object will affect the communication costs. An extension of this abstraction is necessary to facilitate implementations.

In Chapter 5, we extend Johnsson's concept by defining levels of data within an algorithm.

This natural idea allows determination of the communication costs that will exist for an algorithm on a SIMD machine using only high level constructs. Further, we present a strategy which allows these communication costs to be reduced and conclude the section with total communication costs and run times over various sized problems. The benefit of this approach is two-fold. It allows identification of the bottlenecks that will occur in a parallel algorithm and how these may be dealt with, and moreover, it allows extensions to a method to be implemented.

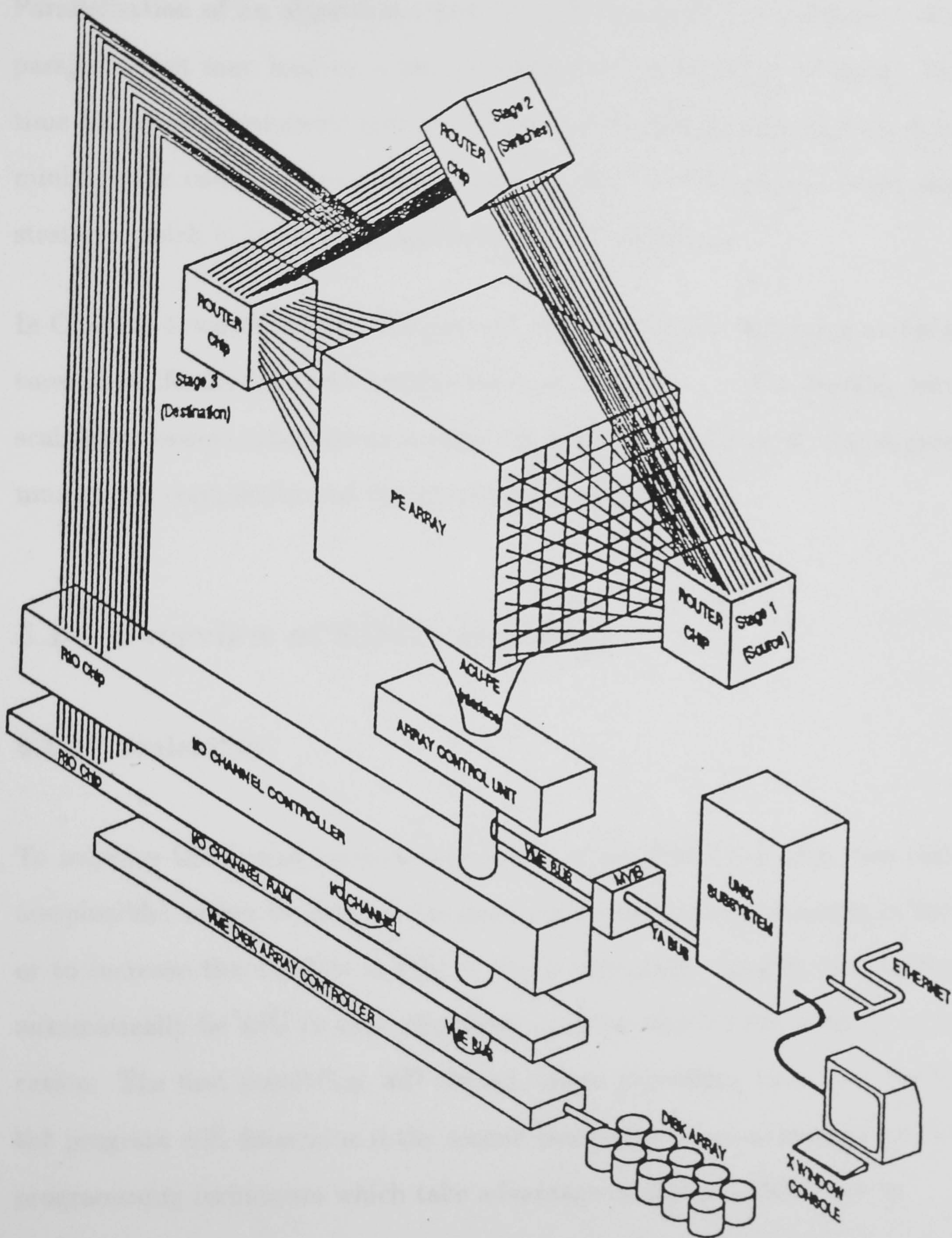


Figure 16: General layout of a MasPar. This figure is referred to in Section 4.4.

5 PARALLELISATION STRATEGY

Parallelisation of an algorithm using the SIMD model is not difficult. However, naive parallelisation may lead to a low processing to communication ratio. The processing time for parallel programs may be reduced by applying a strategy which is designed to minimise the communication that occurs. In this section, we present an implementation strategy, which is in principle applicable to any algorithm.

In Chapter 4, we considered the general problems of load balancing on parallel architectures. Specifically for SIMD architectures, with reference to the MasPar, we now consider scalability issues, reduction of storage via careful application of virtual processing, communication complexity and computational complexity.

5.1 Overview of SIMD problems

5.1.1 Scalability

To improve the overall performance rating of an SIMD machine, two main techniques are possible: either to increase the processing power of the elements in the architecture or to increase the number of PEs in the architecture. Ideally, an implementation will automatically be able to take advantage of these possibilities without requiring modification. The first possibility will always reduce processing time, but the formulation of the program will determine if the second possibility is automatically exploited. General programming techniques which take advantage of this possibility are to

(i) map the major dimensions of arrays that will increase for increased problem size across the PEs, even if the size of these dimensions is much greater than the number of PEs. This leads to the concept of *virtual processors* (VPs), where it is possible to simulate a machine with far more processors than actually exist by using layers of memory in the DPU to simulate more processors. The VP ratio may then be defined as the number

of VPs divided by the number of actual processors. Whilst many SIMD architectures implicitly allow the simulation of VPs, indiscriminatory usage can lead to excessive storage requirements. This is addressed in Section 5.1.2.

(ii) ensure that the size of any loops in the program should not be directly related to changes in the problem size (aside from those which are necessary for defining the VPs). For example, the method by which boundary conditions are imposed should not rely on a loop over the number of boundary conditions, otherwise this loop will take an increasingly disproportionate time compared with the rest of the calculations as the number of boundary conditions increases.

5.1.2 Storage

Using VPs allows a straightforward increase in the size of problems that may be processed. However this may increase storage unnecessarily, because it may not be necessary to store all the values of all arrays. An example of this is a temporary array that is necessary to calculate sections of data for a global array used by the rest of the program. Therefore, a *blocking* technique is used to minimise wastage. In this technique, only the block of temporary data required is calculated and stored. This results in a small loss in readability and requires extra programming, but results in significantly lower overall storage costs. Apart from some small bookkeeping, there is no loss in parallel efficiency.

5.1.3 Communication complexity

Since communication is the major contribution to poor efficiencies of an implementation, it is desirable to ensure that communication requirements are kept as small as possible. Moreover, imposing complex communication pathways may lead to router contention resulting in slow calculations. This is because the DPU must wait until all communication in any one statement has been completed. Therefore it is also important to keep the communication paths simple.

Both of these issues may be demonstrated by the calculation of a matrix-vector (MV) product using a parallel row storage system equivalent to the sequential row storage system discussed in Section 3.9. The only assumption about the matrix itself is that it is square.

Three methods of calculating the MV product are considered. An obvious method is to loop over the rows of the matrix and calculate the dot-product of each row with the vector. This referred to as the row by row method. If the bandwidth is large then this should be a good parallel implementation. However, if the bandwidth is less than the number of processors then not all the processors will be in operation at one time. A variation of this is to compute several row-vector dot products at once. Calculations for this method were done by rearranging data so that a block of eight entries of the MV product were calculated simultaneously, before calculating the next block. This is referred to as the multiple row method.

The third method is to partition the original matrix and then create a second matrix where each row is the transpose of the vector. Then the dot-products of the partitioned matrices and the second matrix are calculated and summed. This is referred to as the partition method.

Table 7 shows the elapsed time (in milliseconds) required to calculate several MV products using each of the methods outlined above. The size of the matrix used in each case is given by n^2 , where n is shown in the first row of the table. For $n = 125$, the partitioned method results in the fastest calculation. The row by row method takes 26 milliseconds longer, whilst the multiple row method takes more than five times as long. As n increases, the elapsed time for the row by row method increases at a much slower rate than the elapsed times for both the multiple row method and the partition method. Elapsed times for $n = 2744$ using the multiple row method could not be obtained as the machine crashed repeatedly before producing a solution. This is possibly because the communication pathways required by this method were complex, or because of storage problems but this has not been investigated.

Clearly the row by row method is the fastest approach for the two bigger problems. The communication overhead of rearranging data, so that the other two methods can be performed, is far too great.

Two additional problems with each of these methods may be identified. Firstly, the ratio of the size of the internal loop of the calculation and n is close to 1. This means that there is little overall parallelisation. Moreover, none of the methods take advantage of the sparsity of the matrices that are produced by the FE method.

5.1.4 Computational complexity

A wealth of powerful commands are available in MPFORTRAN which allow for example reshaping an array from one structure to another. These commands are appealing because they allow concise coding.

In an early implementation to calculate the shape functions, such 'powerful' commands were used (see Mallick and Shaw [60]). However, experience has shown that the shape function data calculations are best achieved by using 'simplistic' commands, namely, hard coding calculations by unrolling arithmetic expressions. Table 8 shows execution rates in MFlops for the six parts of the shape function calculation routine.

We can see that the calculations for the determinant and the inverse of the Jacobian matrices, which had been slowed by the use of powerful commands, now perform far better, simply as a result of constraining the calculations to basic operations.

Whilst all the above considerations are important, the goal of the eventual implementation is to have a readable code which processes a problem as fast as possible. Since the processing of a program on a parallel computer consists of computation time and communication time, we seek to find an implementation that minimises communication.

5.2 Previous SIMD strategies

It has been noted in the literature that problems with the calculation of the MV product on SIMD architectures exist (see Fischer and Patera [61]). The EBE method of Liou and Tezduyar [71] addresses this problem by taking advantage of the following property: the assembled stiffness matrix is only needed so that its action on a known vector may be calculated. Since this matrix is assembled from the matrix addition of previously calculated elemental matrices, equation (71) may be used to avoid constructing the global stiffness matrix:

$$KU = \sum_i [K^{(i)}] \{U^{(i)}\} \quad (71)$$

where K is the global stiffness matrix which is the sum of the i elemental matrices $K^{(i)}$, and U is a known vector which may be decomposed into i vectors $U^{(i)}$.

Imposition of boundary conditions must then be performed on the elemental matrices themselves. This may be performed in a sequential manner, but this affects the scalability of the algorithm as outlined in Section 5.1.1. For this reason, boundary condition imposition is sometimes ignored in the calculations for execution rates (for example, Sawley [65], Sawley and Tegner [68], and Sawley and Bergman [76]).

Whilst the EBE method allows calculations for the MV product to be performed in a more efficient manner, there is no clear framework for how seemingly sequential operations may be performed, nor how extensions to an initial program may be implemented.

Another method used to improve overall performance figures is to fine tune the code in some way peculiar to the machine. Typical techniques are to code computationally intensive parts of the code in a low-level language, or to hardwire communication routes in some optimal fashion as may be done on the Connection Machine (see Tezduyar *et al* [77]). This reduces the portability of the code, but leads to increased performance. However, since these methods cannot be standardised across platforms of similar architecture, they

are not ideal and therefore not used.

5.3 Measuring Communication Costs

Here we consider how the total inter-processor communication (IPC) may be calculated. Communication occurs either via the local router from shifts in data across processors or from the global router. For shift commands which use the local router, evaluating the cost of communication is simple. Assuming unit cost for local router transfer, one may use the *taxicab metric* defined in equation (72).

$$C = \|x_2 - x_1\| + \|y_2 - y_1\| \quad (72)$$

The MasPar however allows X-net communication, thus a suitable metric for this is,

$$C_1 = \|y_2 - y_1\| + \min\|x_1 - (x_2 \pm (y_2 - y_1))\| \quad (73)$$

$$C_2 = \|x_2 - x_1\| + \min\|y_1 - (y_2 \pm (x_2 - x_1))\| \quad (74)$$

$$C = \min(C_1, C_2) \quad (75)$$

For the global router, it is difficult to define an algebraic metric for the cost. Consider the vector addressed statement below:

```
nposn(1:nelem) = nconn(nv,1:nelem)
```

```
unpl(1:nelem)= unpl(1:nelem) + sf(1:nelem,nv)*unew(nposn(1:nelem))
```

Calculating the communication cost for this statement is difficult because the cost of transferring data from one map of arbitrary processors to another depends both on the communication map and the hardware organisation.

For simplicity, we assume that a vector addressed command has unit cost, κ , which represents the time to transfer information from p to p processors where p is the number of processors in the architecture. This assumes that if a bijective mapping exists, then the time for the global router to complete communication is equal for all bijective mappings. We further assume that if an arbitrary mapping may be broken down into a minimum of k bijective mappings, then the resulting communication will take k times longer than the time for a single bijective mapping. Therefore the cost of global router communication C_{global} is

$$C_{global} = k\kappa \quad (76)$$

where k is the minimal number of bijective mappings for the communication pathways required.

It is not clear that the compiler will implement a pathway configuration which matches this value. However, it allows us to form an estimate of the communication costs for vector addressed statements.

5.4 Analysis of finite element algorithm

We now consider implementation of the FE algorithm of Shaw [19], [20]. Figure 17 shows the structure of the algorithm for the sequential version of the method.

There is an initial calculation phase where the data is read, a variety of pointers are calculated and the initial variable field is set. The data intake is a sequential process and cannot be parallelised.

This is followed by a loop over time to resolve the temporal variation with an embedded loop to resolve the non-linearity of the problem. Within these loops, subroutines are used to calculate element matrices and to assemble global matrices for the momentum

equations. Subroutines are again used to find the pressure correction, by calculating and assembling element matrices. Velocity corrections are then calculated. Finally the new velocity and pressure fields is calculated. As described in Section 3.2 the bulk of the calculations occurs in the calculation and assembly of element equations, and the solution of linear systems of equations.

Figure 18 shows psuedo-code for typical sub-calculation. Most of the calculations take place here, and this is where parallelisation must be targeted here.

A basic implementation could involve selecting a simple data structure for each of the sequential array variables, and mapping these according to their largest dimension across the processors. However, this would result in excessive communication, because of data transfer from one set of variables to another. The concept of data levels, discussed in Section 5.5, demonstrates this.

5.5 The Use of Data Levels

In this section, we consider the idea of data levels. This approach is appealing because it demonstrates where communication is going to occur, and permits strategies for avoiding communication costs.

5.5.1 Definition

It is desirable that the data structures of a program intuitively correspond to mathematical quantities of the algorithm. Thus it is possible to define data levels within the algorithm. A data level is defined by the size of the largest dimension of a variable. Thus, any two discretised mathematical quantities whose associated variables have major dimensions of equal size share a data level, even if they represent different physical aspects of a system.

5.5.2 The data level concept applied to an algorithm

An algorithm may be expressed as a sequence of interacting data levels. This sequence may be written as a series of sequential data level movements. It has been found that the most efficient codes are based on very few data levels, which interact very little, and where the interaction within data levels is minimal. This is intuitively appealing. If an algorithm does not have these characteristics then a good approach would be to transform it so these characteristics are present.

To reduce the communication costs of the final program, it is necessary to maximise the inherent parallelisation of the algorithm, without incurring excessive storage costs. Even before programming this may be achieved by modifying the data level movements, via one of the following techniques:

1. Data required for processing may be restructured into a form more suitable for calculation. This idea was originally suggested by Jacobsen [74] who noted that data can be remapped into a more effective form for calculation, and then returned to the original structure, if required.
2. Alternatively it may be preferable to reconsider the data level type and project it entirely into a different data level. This is an extension of Jacobsen's [74] idea. However, this may have implications on the rest of the program, requiring re-evaluation of the data level movements in the program.

Ideally, it would be possible to project all the data levels into just one level, but this is rarely possible. It is more common to select one level as the major working level, and work within this wherever possible.

5.6 Parallelisation strategy

The strategy is presented in the form of a flowchart in Figure 19. There are several steps to the strategy. First, the identification of the data levels in the algorithm, helps to define how data should be structured in the program. Evaluation of the data level movements and calculation of the IPC helps to identify an appropriate choice of elementary object (EO). The EO may also be determined by identifying the largest loops in the algorithm. Then the algorithm is constructed so that wherever possible, each data level is projected into the EO data level. Thus the algorithm is expressed as a sequence of transformations from data level to data level via the EO data level. Where possible, successive data level projections will result in a reduction in the total IPC required. This will result in alterations to the data level movements, necessitating re-evaluation of both data level movements and IPC. However, storage costs may increase as a result of this process, and so total storage costs must be balanced with reasonable IPC. When an acceptable balance is found, the algorithm may be implemented. If this is not possible, then it may be necessary to consider using a different EO, or to modify the algorithm in a way such that the entire process may be re-evaluated, giving a reasonable balance of IPC and storage. If this still results in excessive IPC or unreasonable storage costs, then the target machine is probably unsuitable. Then, a machine with more memory, more processors or a different class of architecture (such as MIMD) may be required.

One common type of loop that may be difficult to parallelise efficiently on SIMD architectures are flow dependent loops. These occur when one or more values of an array are calculated at one stage of the loop and when these are in turn used to calculate subsequent values. Examples of this type of loop are the Gauss-Seidel and SOR methods. Application of the parallelisation strategy to these loops leads to either modification of the algorithm, or use of a different architecture. For the Gauss-Seidel and SOR methods, it is possible to use a red/black form of the algorithm as described in Section 4.5.3. However, for other algorithms which may require flow dependent loops it is sensible to

find an algorithmic modification which no longer requires flow dependent loops. Otherwise, the algorithm will be difficult to implement.

5.7 Application of parallelisation strategy

Here we apply the parallelisation strategy defined in Section 5.6 to the algorithm of Shaw [19], [20].

5.7.1 FE data levels

A generic FE program has four data levels: a *shape function* level (S) (consisting of the shape functions themselves and their derivatives), an *elemental* level (E) (consisting of the connectivity matrix and the element matrices), a *nodal* level (N) (consisting of the mesh geometry and the flow variables at each node), and the *boundary condition* level (B). In addition, we may define the (G), the *global stiffness* level, since this typically large, sparse matrix will require a storage system of some kind, and so cannot be considered a subset of the nodal data level.

5.7.2 Communication costs of original algorithm

After determining the existing data levels, the next step is to calculate the IPC of the algorithm. Values for the IPC of the data level movements of the original algorithm are shown in Table 9. The apostrophe mark indicates that IPC occurs within a data level. It is apparent that only the prediction of the new flow variables require no communication. Every other aspect of the algorithm requires some IPC. Also, the element calculation routines require only a small amount of communication whilst the assembly of the element matrices requires much more.

The communication costs for the imposition of the boundary conditions and the solution

of the linear equations are both directly proportional to a parameter of the problem size. Thus the IPC for these routines increases linearly with each respective parameter. This is more problematic for the linear equations as the number of nodes rapidly increases with problem size. Thus communication will begin to dominate processing time as problem size increases.

5.7.3 Selection of elementary object

The next step of the strategy is to select an EO data level. The large loop in Figure 18, over all the elements represents the largest loop suggesting that the EO data level should be the element data level. However, at the end of a calculation, it is nodal values that are of interest. The decision to work in the element data level and yet obtain results in the nodal level dictates that communication will occur (at the very least) between these two data levels. However, the elemental level is the preferable working data level because of the amount of computation that occurs in it.

We then consider ways of projecting data levels into the element data level. This is done by looking at data level movements and considering them case by case. This is considered in Sections 5.7.4 to 5.7.7.

5.7.4 Case (i): Levels N and S moving to level S ($N, S \rightarrow S$):

Since the shape function data relates to the elements in the mesh, an intuitive move is to project the shape function data level into the elemental data level. This is done by ensuring that the shape function data is mapped such that all the data pertaining to a particular element is mapped to its corresponding VP. Calculations of the shape function information are then done looping over the gauss points. Thus the data level S becomes redundant and the calculations for the shape function data and the element matrices becomes E, $N \rightarrow E$.

5.7.5 Case (ii): Levels E and N moving to level E ($E, N \rightarrow E$):

The problem here is that nodal data is required at element level. This can be resolved by structuring the required nodal data into the elemental form, that is, by transferring the required nodal information onto the processor that requires it. There are two kinds of such data: firstly the geometrical co-ordinate information (required for the shape function calculations) and secondly the run-time flow variable data.

The former can be pre-processed into a new array so that the geometrical data of each node is positioned onto the processor that will need it. This will inevitably require some repetition of data, but results in a reduction of IPC. This extra storage requirement is more than compensated for by the performance improvement.

The latter type of data is the run-time flow variable data, which needs to be transferred once every non-linear iteration. This may be done for all the flow variables at the beginning of each iteration. Though this requires IPC and the creation of extra arrays, the router connections required for this data transfer are the same for each variable. Thus by clustering them together and using the *router_opt* compiler option (which keeps the processor routing connections open for series of similarly routed commands), this becomes an efficient process requiring less than 60ms per non-linear iteration per block of 4096 elements. This is a loss that is more than justified by the savings in time in the calculations for the element matrices. This early restructuring and clustering technique has improved the speed of these calculations by just under 15 MFlops for a single block of elements.

5.7.6 Case (iii): Levels G and N (with IPC) moving to level N ($G, N' \rightarrow N$):

The next major set of calculations that are those that solve the systems of linear equations. Our scheme uses iterative solvers to evaluate solutions and, in this study, the conjugate gradient (CG) solver with diagonal preconditioning (PCG) (see Chin *et al*

[58]) and the conjugate gradient squared (CGS) (see Howard *et al* [55]) algorithm have been used. Iterative solvers are dominated by the evaluation of MV products. This is obvious because, for example, the CG algorithm requires seven vector multiplications and one MV product per iteration. Whilst the vector multiplications may be performed without communication, MV products inevitably involve IPC. The parallelisation of a MV product is highly dependent on the manner in which the matrix is stored. The original skyline scheme used in our sequential code is wholly inappropriate for SIMD machines.

An alternative is to take the MV product from the nodal level into the elemental level, by avoiding assembly of the global stiffness matrix. This is similar to the EBE method (see Liou and Tezduyar [71]) discussed in Section 5.2. Fischer and Patera [61] describe a scatter-gather process which allows the a single matrix-free MV product to be evaluated. This is described below.

Step 1: SCATTER: Broadcast the vector across the processors where required

Step 2: PROCESS: Calculate the elemental MV products

Step 3: GATHER: Assemble the elemental MV products

This method is much faster than any of the methods considered in Section 5.1.3. and has the desirable feature of making the loop required to perform the calculation equal to the number of nodes on each element, as opposed to the total number of nodes. This results in a high degree of processor activity with no IPC during the processing phase. Moreover the assembling of the matrices becomes redundant.

Table 10 shows typical times for a single MV product calculation using this technique. In this table, the communication costs are defined as the time for the scatter and gather operations as a percentage of the total time for the MV product.

Whilst the communication costs seem high, it should be borne in mind that communi-

cation in some form is inevitable for the MV product. Overall, it is the time of the total calculation relative to the machine's practical peak performance that is important, even if this involves an apparently high percentage of communication. Comparing the times for the total process in Table 10 with Table 7, we can see that there has been a dramatic reduction in the time required for calculation, even though there has been an increase in the size of the problems.

During the solution of a linear system of equations, the global stiffness matrix does not change. Thus, only the new solution vector needs to be rebroadcast during successive iterations.

5.7.7 Case (iv): Boundary conditions: Level B moving to level N ($B \rightarrow N$):

A consequence of using the matrix-free technique is that the boundary conditions can no longer be imposed in the normal fashion, since the global stiffness matrix is not assembled. Instead, the boundary conditions must be imposed on every elemental matrix, changing the data level specification from $B \rightarrow N$ to $B \rightarrow E$. This could be performed by using a loop over the number of boundary conditions, imposing each condition in a sequential manner. However, this would be extremely slow. An alternative is project the boundary condition data level into the element data level. This is done by placing the elemental boundary condition data onto the processors that hold data for elements on the boundary. Actual imposition may then be performed using a masking and value array, resulting in a data level specification of $E \rightarrow E$.

5.8 Final data level movements and communication costs

Applications of the parallelisation strategy on the algorithm of Shaw [19], [20] results in data level movements shown in Table 11. Communication during runtime occurs once every non-linear iteration whilst restructuring flow data, and during the scatter-gather of the iterative solvers. Thus the total communication cost for the algorithm per non-linear

iteration is:

$$C = [8\gamma_e\sigma\kappa] + 2[N\gamma_n\sigma k\kappa] \quad (77)$$

where γ_e , γ_n are the number of elemental and nodal blocks respectively, σ is the maximal number of nodes per element, k is the number of MV products in the iterative solver, and N is the number of iterations that the iterative solver performs. Note that one element block and one node block consists of p elements and p nodes respectively, where p is the number of processors in the DPU.

It can be seen that many of the run-time calculations do not require IPC. The only run-time IPC that occurs involve the redistribution of data across processors to facilitate high speed calculations. In the case of restructuring the flow variable data, this time is negligible compared to the overall calculation time. Whilst this cannot be said for the scatter and gather operations involved in calculating the MV product, this technique involves no IPC during the actual calculation phase, and hence a fast MV product calculation.

The resulting structure of the main program is given in Figure 20. Comparing this to the structure of the sequential code of Figure 17, it can be seen that there appears to be very little difference in overall structure. All that has changed is that at the beginning of each non-linear iteration, some flow variable data is restructured. However, when we look at a typical sub-calculation as shown in Figure 21 and compare this with Figure 18 we can see that there is no longer any assembly of the elemental matrices. Moreover, the loop over the elements has been removed. This is no surprise since parallelisation has been targetted here, and we have in a sense taken this loop into the gauss point loop. In addition, there is a loop over the number of blocks of 4096 elements. This is due to the blocking approach discussed in Section 5.1.2 which reduces the memory required by temporary variables.

The actual computer code for the calculation of the left hand side element matrices for

the u-momentum equation is shown in Figures 22 to 27. Lines 11 to 17 give descriptions of the variable names. Lines 19 to 30 define local variables, whilst lines 36 to 43 ensure that arrays are stored on the DPU and mapped correctly across the processors. Lines 46 to 67 define an interface to the external subroutine called shape8 which calculates the shape functions. Lines 91 to 103 begin the loop over the number of element blocks, and define pointers to the arrays to ensure that each subsequent calculation requires the minimum amount of memory. This was discussed in Section 5.1.2. The rest of the code shows the loop over the gauss points (lines 105 to 110) and subsequent calculation of the element matrices (lines 134 to 152).

5.9 Results of Implementation

Application of the parallelisation methodology allows fast calculations of the MV products and a performance increase in the speed of the iterative solvers. For any sized matrix problem, the time for one iteration of the iterative solvers is approximately $CGS = SOR = 2 * PCG$. This is entirely explained by the two MV products per iteration in the CGS and SOR routines, as opposed to one in PCG. In fact, SOR is very slightly lower than CGS, due to slightly more expressions in the CGS solver.

By calculating the number of floating point operations in each subroutine and timing each routine in turn, it is possible to calculate MFlop statistics for each part of the program. This has been done for two three-dimensional cases described below. These statistics are shown in Table 12, along with the percentage of total time spent in each part of the program (excluding pre- and post-processing time).

Case 1: 4913 node, 4096 element driven cavity problem

Case 2: 19683 node, 17576 element driven cavity problem

Clearly the element matrices are calculated very quickly; this is to be expected since

the main parts of the calculations require no IPC after the flow variable data has been restructured into the elemental data level. Boundary condition imposition appears to perform badly - this is because no useful work is performed by the processors storing elements that do not require any boundary conditions. However, the method requires no IPC, and hence takes only a small percentage of the total time for calculation.

The iterative solvers combined take up over 40% of the total calculation time, and run at under 10MFlops. This is entirely due to the communication required in each step of the MV product calculation. PCG performs slightly better than CGS because there is only one such calculation that needs to be performed in each PCG iteration as opposed to two per iteration in CGS.

5.10 Discussion

Application of the parallelisation strategy presented in Figure 19 has resulted in a code where very little communication occurs. This has been mainly due to the understanding of data level structures inherent within the code, and the flexibility with which they can be modified into other data levels. This approach has been successful on all parts except for the iterative solvers where it is not possible to eliminate the communication that results from calculating the MV product. For this reason, explicit codes have been more popular than implicit ones, but since they require a much smaller time step this does not necessarily make them the method of choice for solving flow problems. Moreover, since the paths required for this communication are always the same, it is possible to fine-tune the code and use lower level languages to minimise the overall time for the communications that are involved; for example, users of the Connection Machine have benefited from vendor-supplied communication pathway codes.

The distinction between the EBE method and this strategy is subtle. The EBE method identifies that the MV product is a problem because it is not possible to calculate the MV product efficiently using an assembled storage scheme. Therefore, it instead uses

equation (71).

Our approach on the other hand gives a framework for the parallelisation of any algorithm which is a reasonable candidate for SIMD programming. As a consequence of this, equation (71) is used because it simplifies data level movements. Moreover, the calculation of the boundary conditions immediately fall into place as a non-sequential operation, and provides a framework for extensions. Whilst application of the strategy on other algorithms is beyond the scope of this thesis, the approach is demonstrated in Chapter 6 on an algorithmic modification of the FE method.

In assessing the performance of a parallel code, it is common to quote on overall parallelisation efficiencies; this is more usually reported when using MIMD architectures where it is possible to attain a high efficiency by process parallelism. With SIMD computers however, it is not usual to obtain high efficiencies, and instead MFlop statistics are usually given as an indication of performance, but these figures have to be considered along with the peak performances of machines to give a fair comparative measure of the performance of strategy employed. Moreover there are often architectural differences between families of machines which means that direct comparisons are not entirely valid. Nevertheless, Tezduyar *et al* [77] using an implicit finite element scheme reports a performance of 2 GFlops on 1K nodes of a CM-200 (using 64-bit arithmetic). This machine has a peak performance of 10 GFlops (when using 64-bit arithmetic) representing a performance of 20%, which is comparable to our performance even though we have used only high level language constructs.

Moreover, the processing time for the implementation scales with problem size. This is demonstrated by measuring the elapsed time for a series of different problems. Figure 28 shows the time elapsed time per non-linear iteration for various sized problems. Once the number of elements in the problem exceeds the number of processors in the DPU, the elapsed time is proportional to the number of elements. This is not true for small problems since the DPU is not effectively utilised. Note that in this figure, the PCG method was used using 25 iterations for the solution of systems of linear equations generated from the

momentum equations, and 50 iterations for those generated from the pressure correction equation.

In practice, the number of iterations necessary for increasing mesh size usually increases. However, this figure shows clearly that the solution time increases proportionally to the size of the problem, indicating that the solution scheme is scaleable with respect to problem size. However, this figure tells us nothing about the effect of an increase in the number of iterations of the iterative solver.

In practice however, the number of iterations required for the linear equation solver is a function of the number of nodes and the Reynolds number. This is because matrices generated become less diagonally dominant as the Reynolds number increases, affecting the convergent properties of the linear solver. However, the time per iteration of the linear solvers also scales linearly with problem size. This is shown in Figure 29 where the PCG solver is used.

Thus in practice, the time per non-linear iteration will not be proportional to the problem size, because bigger problems require more iterations of the linear solver, in addition to the extra work in the rest of the calculations. Nevertheless, the implementation is automatically able to exploit both increases in processing power of each processor, and increases in the number of processors. Whilst the MP1104 used in this work has only 4096 processors, a fully configured MasPar has 16384 processors. Given that a fully configured MP2216 can store 64 Kbytes per processor and operate at 6300 MFlops (see Brauni [62]), this suggests that the current implementation would be able to solve problems with over 300,000 degrees of freedom at a rate of over 1200 MFlops. However, this is beyond the scope of this work and has not been attempted.

	n=125	n=729	n=2744
Row by row method	115 ms	680 ms	2750 ms
Multiple rows method	452 ms	11930 ms	n/a
Partition method	89 ms	1898 ms	19007 ms

Table 7: Elapsed times (in milliseconds) for the calculation of a single MV product. The MV products are calculated using the three different methods discussed in Section 5.1.3. Here, n gives the number of rows of the square matrix used in the calculation. This table is also referred to in Section 5.7.6.

Calculation of:	MFlops using 'powerful command' strategy	MFlops using 'simplistic command strategy
Interpolation values	12.0	12.0
Local derivatives	47.6	47.6
Jacobian matrices	11.46	11.46
Determinant	0.32	67.0
Inverse of Jacobian	0.908	49.1
Global derivatives	11.8	11.8

Table 8: Processing speeds for the different sections of the shape function calculations. This is discussed in Section 5.1.4.

Costs	Movement	IPC per non-linear iteration
Shape functions	$N, S \rightarrow S$	-
Calculation of element matrices of u, v, w	$N, E \rightarrow E$	72κ
Assembly of element matrices of u, v, w	$E \rightarrow G$	792κ
Calculation of element matrices of p	$N, E \rightarrow E$	24κ
Assembly of element matrices of p	$E \rightarrow G$	264κ
Imposition of boundary conditions	$B \rightarrow G$	\propto (no. bcs)
Solution of linear equations	$G, N' \rightarrow N$	\propto (no. nodes)
Updating flow variables	$N \rightarrow N$	0

Table 9: Data level movements and IPC values of the original algorithm, referred to in Section 5.7.2. Here, the apostrophe mark indicates that IPC occurs within a data level. Also, the IPC for the shape functions is not given since this subroutine is included in the elemental matrix calculations.

	$n = 4096$	$n = 19683$
Scatter	8.3 ms	42.4 ms
Process	6.5 ms	51.1 ms
Gather	17.2 ms	88.8 ms
Communication costs	80%	72%
Total:	32.0 ms	182.3 ms

Table 10: Elapsed times (in milliseconds) and communication costs for a single MV product calculation using the scatter-gather approach of Fischer and Patera [61]. Here, n refers to the size of the vector in the product. The communication costs are defined as the time for the scatter and gather operations as a percentage of the total time for the MV product. This is discussed in Section 5.7.6.

Routine	Movement	IPC per non-linear iteration
Preprocessing:		
Restructure nodal co-ordinate data	$N \rightarrow E$	$3\sigma\gamma_e\kappa$
Restructure boundary condition data	$B \rightarrow E$	$\propto(\text{no. of bcs})$
Run-time:		
Restructure flow variable data	$N \rightarrow E$	$8\sigma\gamma_e\kappa$
Calculate shape functions	$E \rightarrow E$	0
Calculate element matrices	$E \rightarrow E$	0
Impose boundary conditions	$E \rightarrow E$	0
Solve linear equations (Scatter)	$N \rightarrow E$	$Nk\sigma\gamma_n\kappa$
Solve linear equations (Calculate)	$E \rightarrow E$	0
Solve linear equations (Gather)	$E \rightarrow N$	$Nk\sigma\gamma_n\kappa$
Update flow variables	$N \rightarrow N$	0

Table 11: Data level movements and communication costs for final algorithm. See Section 5.8 for details of the variables used in the IPC column.

	Case 1		Case 2	
Routine	MFlops	percentage	MFlops	percentage
		time		time
Calculate shape functions	18.2	-	20.3	-
Calculate lhs element matrices	54.2	35.2	54.2	39.3
Calculate rhs element matrices	35.6	15.0	39.5	15.9
Impose boundary conditions	3.8	1.2	2.3	1.5
CGS iterative solver	6.1	34.1	7.9	31.8
PCG iterative solver	6.7	11.6	8.8	10.6
Reading and writing data	-	2.9	-	0.9
Speed for main program	23.2		28.2	

Table 12: MFlop rates and execution times (as a percentage of total execution time) for each part of the final algorithm. The time for the shape functions is not given as it is included in the time for calculation of element matrices. Note also that the speed for the main program does not include the time for reading and writing data. This table is discussed in Section 5.9.

Main Program

Read data

Set pointers to arrays

Set initial field values

LOOP for time steps

 LOOP for non-linearity

 Calculate left hand side element

 matrices from momentum equations
 and assemble

 Calculate right hand side element

 matrices from momentum equations
 and assemble

 Calculate element matrices for pressure

 correction, assemble and solve

 Calculate velocity corrections

 Calculate new velocity and pressure components

 ENDLOOP for non-linearity

 Update new variables from old variables

ENDLOOP for time steps

Write data

End main program

Figure 17: Structure of the main program of the sequential implementation. This is referred to in Sections 5.4 and 5.8.

Subroutine to calculate one velocity/pressure component

 LOOP over all elements

 LOOP over Gauss points

 Calculate shape functions and their derivatives

 Calculate element matrix

 ENDLOOP over Gauss points

 Assemble into global matrix

 ENDLOOP over elements

 Impose fixed-value boundary conditions

 Find solution to linear simultaneous equations

 Predict new velocities/pressures

End subroutine

Figure 18: Typical structure of a sub-calculation of the sequential implementation. This is referred to in Sections 5.4. and 5.8.

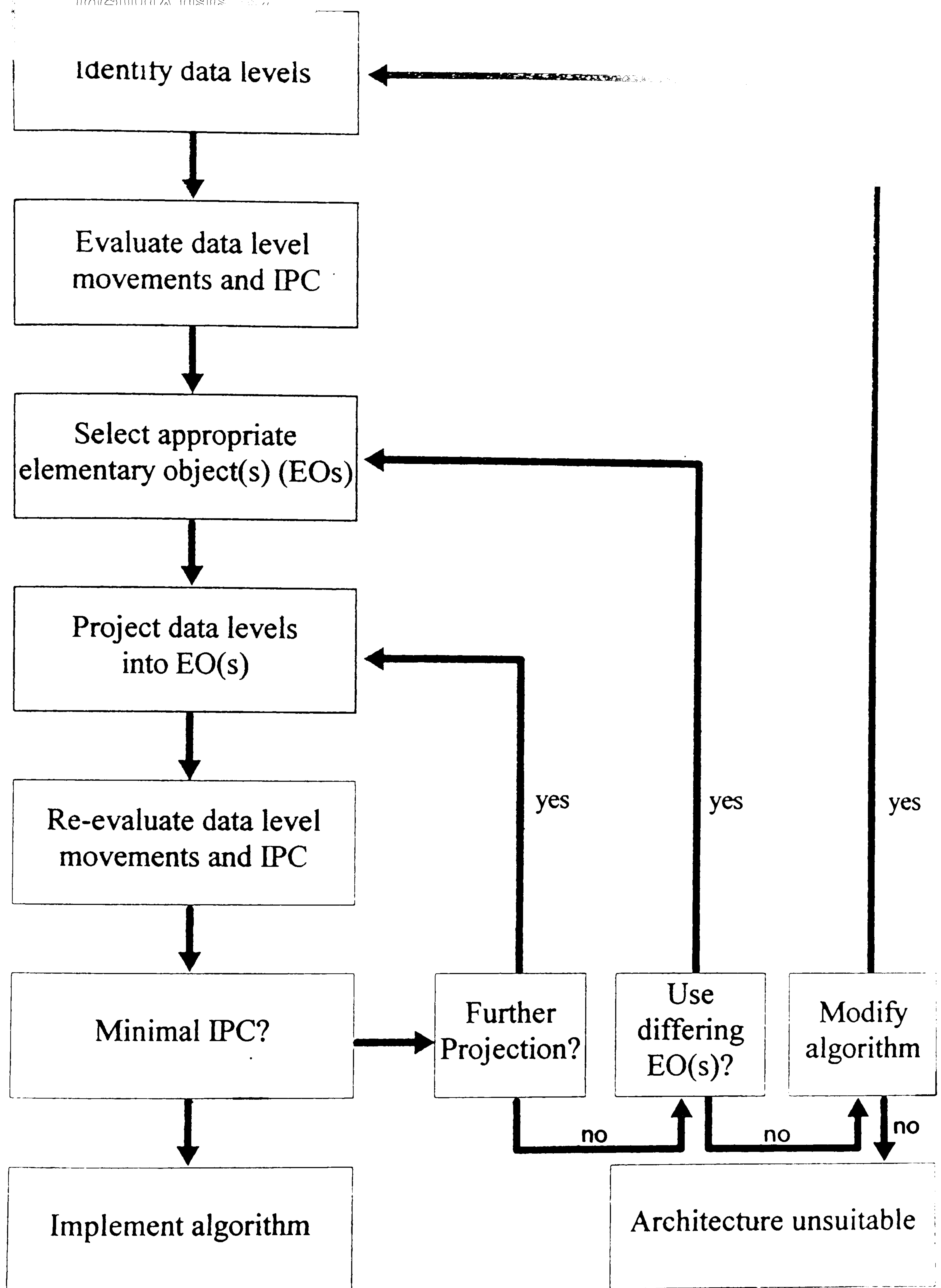


Figure 19: Strategy for parallelisation. This is discussed in Section 5.6 and referred to in Section 5.10.

Main Program

Read data

Set pointers to arrays

Set initial field values

LOOP for time steps

 LOOP for non-linearity

Restructure flow variable data

 Calculate left hand side element

 matrices from momentum equations

 Calculate right hand side element

 matrices from momentum equations,

 and solve

 Calculate element matrices for pressure

 correction and solve

 Calculate velocity corrections

 Calculate new velocity and pressure components

 ENDLOOP for non-linearity

 Update new variables from old variables

ENDLOOP for time steps

Write data

End main program

Figure 20: Structure of the main program of the parallel implementation. This is discussed in Section 5.8.

Subroutine to calculate one velocity/pressure component

 LOOP over the element blocks

 LOOP over Gauss points

 Calculate shape functions and their derivatives

 Calculate element matrix

 ENDLOOP over Gauss points

 ENDLOOP over element blocks

Impose fixed-value boundary conditions

Find solution to linear simultaneous equations

Predict new velocities/pressures

End subroutine

Figure 21: Typical structure of a sub-calculation of the parallel implementation. This is discussed in Section 5.8.


```

1      subroutine elqlhs
2
3      C
4      C      DESCRIPTION - CALCULATE THE LHS ELEMENT MATRICES FOR THE
5      C                        MOMENTUM EQUATIONS
6      C      PROGRAMMER - SWAPAN MALLICK
7      C
8
9
10     C      SET UP VARIABLES.
11     C      NPP      : NUMBER OF PROCESSORS PRESENT (=4096)
12     C      NPE      : NUMBER OF NODES PER ELEMENT
13     C      NELEM    : TOTAL NUMBER OF ELEMENTS
14     C      NBELEM   : NUMBER OF BLOCKS OF 4096 ELEMENTS
15     C      NGP      : NUMBER OF GAUSS QUADRATURE POINTS
16     C                : IN EACH CARTESIAN DIRECTION
17     C      DT       : TIME STEP
18
19     include 'common.f'
20
21     integer :: npe,nv,ni,nj,nk,ib,ibs,ibf, ibe,npp
22     real    :: dt,xi,eta,zeta,temp
23     real,    dimension(npp,3,8)    :: gdsf
24     real,    dimension(npp,8)      :: sf
25     real,    dimension(npp,8,8)    :: amass
26     real,    dimension(4,4)        :: gauss,wt
27     real,    dimension(npp)        :: det
28     real,    dimension(npp)        :: const
29     real,    dimension(npp)        :: unpf,vnpf,wnpf

```

Figure 22: Part 1 of parallel subroutine code. This is discussed in Section 5.8.

```

30      integer, dimension(npp)      :: ntem
31
32  C
33  C      MAP VARIABLES TO THE DPU, WITH MAJOR DIMENSION
34  C      ACROSS THE PROCESSORS
35
36  CMPF  ONDPU gdsf
37  CMPF  ONDPU sf
38  CMPF  ONDPU amass
39  CMPF  ONDPU det
40  CMPF  ONDPU const
41  CMPF  MAP gdsf(ALLBITS,memory,memory)
42  CMPF  MAP sf(ALLBITS,memory)
43  CMPF  MAP amass(ALLBITS,memory,memory)
44
45
46  C
47  C      DEFINE INTERFACE TO SHAPE FUNCTION SUBROUTINE
48  C
49
50
51      interface
52          subroutine shape8(xi,eta,zeta,sf,gdsf,det,elxyz,nelem,
53              $              npelem,ib,npp)
54              integer :: nelem,npelem,ib,npp
55              real    :: xi, eta, zeta
56              real, dimension(npp,8)      :: sf
57              real, dimension(npp,3,8)    :: gdsf
58              real, dimension(npp)        :: det

```

Figure 23: Part 2 of parallel subroutine code. This is discussed in Section 5.8.


```

59          real, dimension(npelem,3,8) :: elxyz
60  CMPF  ONDPU gdsf
61  CMPF  ONDPU sf
62  CMPF  ONDPU det
63  CMPF  MAP gdsf(ALLBITS,memory,memory)
64  CMPF  MAP sf(ALLBITS,memory)
65  CMPF  MAP elxyz(ALLBITS,memory,memory)
66          end subroutine
67  end interface
68
69
70  C
71  C      SET UP ARRAYS FOR QUADRATURE
72  C
73
74      data gauss/4*0.0d0,-.57735027d0,.57735027d0,2*0.0d0,-.77459667d0,
75      $ 0.0d0,.77459667d0,0.0d0,-.86113631d0,
76      $ -.33998104d0,.33998104d0,.86113631d0/
77
78      data wt/2.0d0,3*0.0d0,2*1.0d0,2*0.0d0,.55555555d0,.88888888d0,
79      $ .55555555d0,0.0d0,.34785485d0,2*.65214515d0,.34785485d0/
80
81
82  C
83  C      INITIALISE ARRAYS
84  C
85      elhs=0.0
86
87  C

```

Figure 24: Part 3 of parallel subroutine code. This is discussed in Section 5.8.

```

88  C      LOOP OVER THE NUMBER OF BLOCKS OF ELEMENTS
89  C
90
91      do 50 ib=1,nbelem
92          amass=0.0
93
94  C
95  C      DEFINE POINTERS FOR ARRAYS FOR BLOCK NUMBER IB
96  C      IBS : START POINTER IN ARRAY
97  C      IBE : END POINTER IN ARRAY
98  C
99      ibs=((ib-1)*npp)+1
100     ibf=ibs+npp-1
101     if(ibf .gt. nelem)ibf=nelem
102
103     ibe=ibf-ibs+1      ! define the end for arrays
104
105  C
106  C      DO-LOOPS ON NUMERICAL (GAUSS) QUADRATURE BEGIN HERE
107  C
108     do 100 ni=1,ngp
109     do 100 nj=1,ngp
110     do 100 nk=1,ngp
111         xi=gauss(ni,ngp)
112         eta=gauss(nj,ngp)
113         zeta=gauss(nk,ngp)
114
115  C
116  C      CALCULATE THE SHAPE FUNCTIONS

```

Figure 25: Part 4 of parallel subroutine code. This is discussed in Section 5.8.


```

117  C
118      call shape8(xi,eta,zeta,sf,gdsf,det,elxyz,
119      1      nelem,npelem,ib,npp)
120      const(1:ibe)=det(1:ibe)*wt(ni,ngp)*wt(nj,ngp)*wt(nk,ngp)
121
122      unp1=0.0
123      vnp1=0.0
124      wnp1=0.0
125
126      do 900 nv=1,npe
127          ntem(1:ibe)=nconn(nv,ibs:ibf)
128          unp1(1:ibe)=unp1(1:ibe)+sf(1:ibe,nv)*unew(ntem(1:ibe))
129          vnp1(1:ibe)=vnp1(1:ibe)+sf(1:ibe,nv)*vnew(ntem(1:ibe))
130          wnp1(1:ibe)=wnp1(1:ibe)+sf(1:ibe,nv)*wnew(ntem(1:ibe))
131      900      continue
132
133
134  C
135  C      CALCULATE THE LHS COMPONENTS
136  C
137      do 80 i=1,npe
138      60      do 85 j=1,npe
139
140          elhs(ibs:ibf,i,j)=elhs(ibs:ibf,i,j)+const(1:ibe)* (
141      1      dt*rho*theta*unp1(1:ibe)*sf(1:ibe,i)*gdsf(1:ibe,1,j)
142      2      +dt*rho*theta*vnp1(1:ibe)*sf(1:ibe,i)*gdsf(1:ibe,2,j)
143      3      +dt*rho*theta*wnp1(1:ibe)*sf(1:ibe,i)*gdsf(1:ibe,3,j)
144      4      +dt*theta*amu*gdsf(1:ibe,1,i)*gdsf(1:ibe,1,j)
145      5      +dt*theta*amu*gdsf(1:ibe,2,i)*gdsf(1:ibe,2,j)

```

Figure 26: Part 5 of parallel subroutine code. This is discussed in Section 5.8.

```

146      6      +dt*theta*amu*gdsf(1:ibe,3,i)*gdsf(1:ibe,3,j))
147
148      amass(1:ibe,i,j)=amass(1:ibe,i,j)+const(1:ibe)*rho*
149      1      sf(1:ibe,i)*sf(1:ibe,j)
150
151      85      continue
152      80      continue
153      100 continue
154
155 C
156 C      STORE THE DIAGONALS OF THE LHS
157 C
158      do 300 i=1,npe
159      aii(i,ibs:ibf)=elhs(ibs:ibf,i,i)
160      300      continue
161
162 C
163 C      ADD MASS TERMS TO THE LHS
164 C
165      elhs(ibs:ibf,1:8,1:8)=elhs(ibs:ibf,1:8,1:8)+amass(1:ibe,1:8,1:8)
166
167 C
168 C      PROCEED TO NEXT BLOCK OF ELEMENTS
169 C
170      50 continue
171
172      return
173      end

```

Figure 27: Part 6 of parallel subroutine code. This is discussed in Section 5.8.

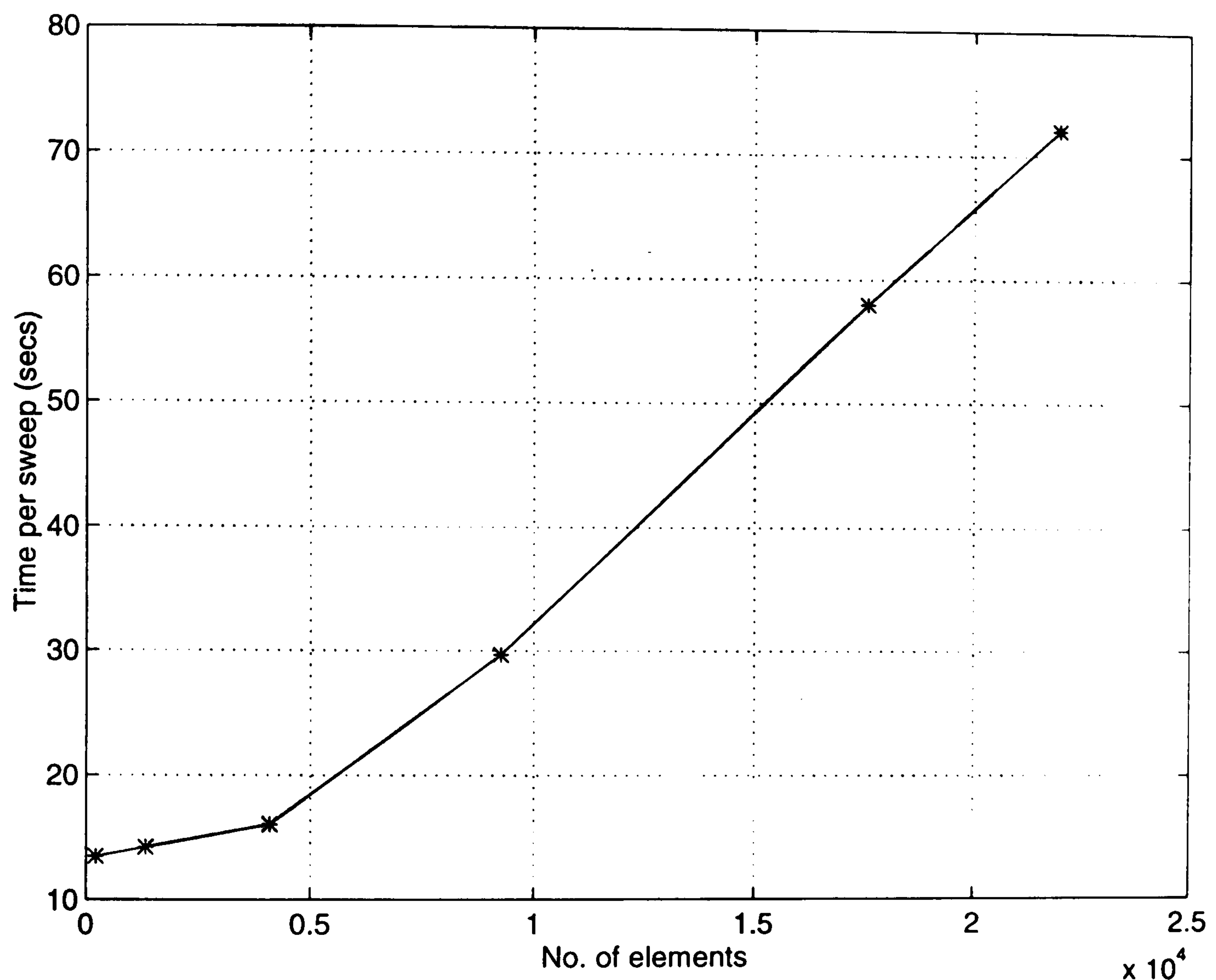


Figure 28: Solution times for various sized problems. The time (in seconds) per non-linear iteration is measured as a function of the number of elements involving a variety of problems. In each case, each non-linear iteration involved 25 iterations of PCG if matrices were generated from the momentum equations, and 50 iterations if matrices were generated from the pressure correction equation. This is further discussed in Section 5.10.

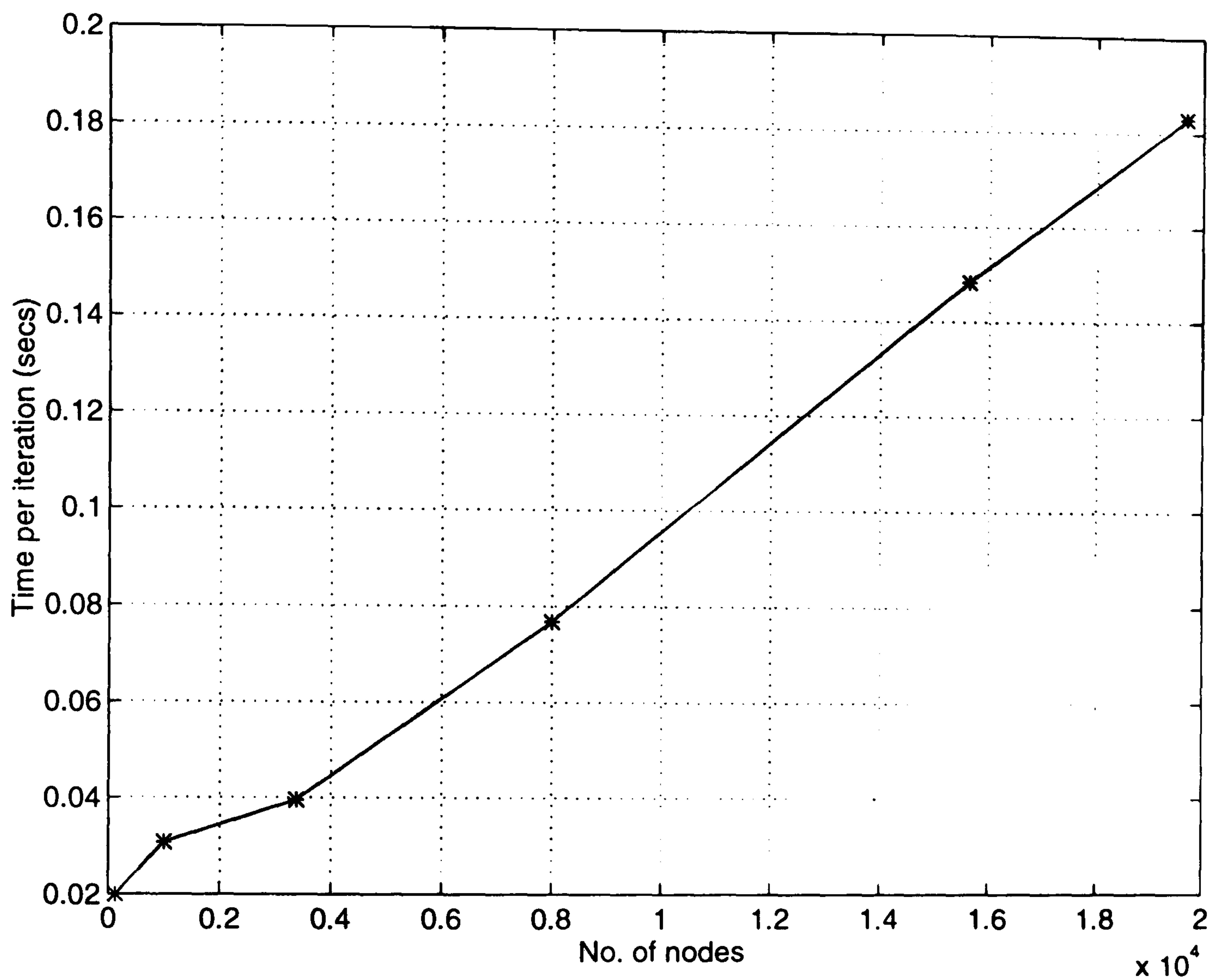


Figure 29: Solution times for iterative solvers on various sized matrices. The time for one iteration of the PCG solver is measured as a function of the number of nodes. This is discussed in Section 5.10.

6 APPLICATION OF PARALLEL STRATEGY

6.1 Introduction

In Section 2.7 a method for solving the incompressible Navier-Stokes equations was presented. However, generating solutions becomes problematic as the Reynolds number increases. Therefore some kind of upwinding technique is typically used to discretise the convective term to allow reasonable solutions to be generated. This is discussed in Section 6.2. Parallelisation of the upwinding technique is awkward because of the nature of the calculations involved. However, the strategy for parallelisation presented in Section 5.6 which provided a framework for parallelisation also caters for extensions. This is discussed in Sections 6.5 and 6.6. Results for the overall implementation are discussed in Section 6.7.

6.2 Limitations of Incompressible Codes

Discretisation of the Navier-Stokes equations have been considered in Chapter 2 for flows at low Reynolds numbers. However, as the Reynolds number increases, the *Peclet* number (Pe), also known as the local cell Reynolds number, also increases. Above some critical value, which is dependent on the characteristic length and velocity, solution schemes may exhibit difficulties in converging, and oscillations in a solution may be found (see Zienkiewicz and Taylor [31]). Whilst it may be possible to reduce the Peclet number in certain regions by mesh refinement, it may not always be clear exactly where the Peclet number is causing a mesh convergence problem.

Methods were discussed in Section 2.4.4 of how to discretise the convection term.

Of these methods, the UPG method seems to be the most appropriate, because it is intuitively simple, seems the most natural method available. Moreover, the computational work required for this method is relatively low compared to the higher order methods.

6.3 Upwind Petrov Galerkin Method

In the UPG method of Kelly *et al* [78], the Galerkin technique of selecting the weight functions (W_i) of each element to equal the approximation functions (N_i) of the primitive variables is modified by equation (78),

$$W_i = N_i + \alpha \tilde{W}_i, \quad (78)$$

where the additional term $\alpha \tilde{W}_i$ may be defined by equation (79),

$$\alpha \tilde{W}_i = \alpha \frac{h}{2} \frac{\partial N_i}{\partial x} (\text{sign} A), \quad (79)$$

where A and h are representative velocity and length scales of the element.

For steady-state convection-diffusion problems in one dimension, Zienkiewicz and Taylor [31] show that if the value of α is defined by equation (80), then exact nodal values will be given for all values of Pe .

$$\|\alpha\| = \alpha_{opt} = \coth h \|Pe\| - \frac{1}{\|Pe\|}, \quad (80)$$

where Pe is defined by equation (81).

$$Pe = \frac{Ah}{2\nu}. \quad (81)$$

In two dimensions however, the upwinding must be directional. Since the convection term is only active in the direction of the resultant element velocity, a corrective term (\tilde{W}_i) should have no non-zero coefficient except in the direction of the resultant velocity. This is possible using the correction factor introduced by Kelly *et al* [78], given in equation

(82),

$$W_k = N_k + \alpha \tilde{W}_k = N_k + \frac{\alpha h}{2} \frac{A_i}{\|A\|} \frac{\partial N_k}{\partial x_i}. \quad (82)$$

where α may be defined by equation (80), and with Pe defined by equation (83),

$$Pe = \frac{\|A\| h}{2\nu} \quad (83)$$

where

$$\|A\| = (A_x^2 + A_y^2)^{\frac{1}{2}}. \quad (84)$$

6.4 Implementation Details

We have implemented the SUPG algorithm on a sequential architecture in the following manner:

1. Compute the centroidal position
2. Compute the centroidal velocity
3. Calculate the characteristic length
4. Define Pe and α_{opt}
5. Modify existing weight functions

For step (1), calculation of the centroidal position in two dimensions may be carried out as follows.

For a given quadrilateral ABCD, as shown in Figure 30, with vertices at position vectors a, b, c, d (with respect to some arbitrary origin), two triangles ABC, and ACD may be defined. Since the centroid of a general triangle with vertices at position vectors p, q, r is given by $\frac{p+q+r}{3}$, and because the centroid (\bar{x}) of a composite body made up of i parts can be determined from the weights (w_i) and centroids (\bar{x}_i) of the individual bodies by equation (85),

$$\bar{x} = \frac{\sum_i w_i \bar{x}_i}{\sum_i w_i}, \quad (85)$$

then the centroid of ABCD may be defined as

$$C = k_1 \frac{1}{3}(a + b + c) + k_2 \frac{1}{3}(a + c + d), \quad (86)$$

where

$$k_1 = \frac{1}{2} \|(c - a) \times (b - a)\|, \quad (87)$$

$$k_2 = \frac{1}{2} \|(c - a) \times (d - a)\|. \quad (88)$$

This calculation may be extended into three dimensions, by considering an arbitrary hexahedral to be made up of tetrahedra. If the element is cuboidal, then the centroid may be simply defined as $\frac{1}{6} \sum_{i=1}^6 x_i$, $\frac{1}{6} \sum_{i=1}^6 y_i$, $\frac{1}{6} \sum_{i=1}^6 z_i$. For the meshes used in this work, all elements have been regular cuboids, so this definition for the centroid has been used.

For arbitrarily hexahedral elements however, it may be necessary to accurately determine the position of the centroid. Consider the hexahedral element shown in Figure 31. This has been divided into six pyramids by choosing an arbitrary point inside the element.

such as the mean of the co-ordinates of the vertex points. Each pyramid may be divided into two tetrahedra (not shown in the figure), so that for example, pyramid ABCDP is split into ABDP and BDCP. Since the centroid of a tetrahedra is easily determined, as is its volume, equation (85) may be used to evaluate the exact position of the centroid of the hexahedra.

For step (2), the centroidal velocities may be calculated directly from the interpolation functions. It is necessary that the velocity components A_x and A_y in a particular element are substantially constant. With regard to the algorithm of Shaw [19], [20], since transient solutions are obtained by time stepping, the primitives in any one time slice will be almost constant if the solution scheme is converging, thereby satisfying this criterion.

For step (3), Zienkiewicz and Taylor [31] note that the choice of h is somewhat arbitrary, but must be reasonably well defined. Obtaining h in two dimensions is not difficult for bi-linear elements. However, for tri-linear grids in three dimensions, this is more problematic since each face of a hexahedral element consists of four points which do not necessarily lie in a plane. In order to obtain a reasonable estimate, the approach used is to divide each face into two planes and calculate the distance from the intersections of these.

The remainder of the procedure, that is steps (4) and (5) follow simply from equations (80), (83) and (82).

6.5 Problems of Parallelising Upwinding

Parallelising the upwinding technique however is non-trivial. Using the data level concept defined in Chapter 5, and the steps of the calculation defined in Section 6.4, the data level movements may be defined. It will be shown that there is an interaction between the nodal and elemental data levels and this needs to be resolved in order to allow the upwinding calculations to occur successfully.

For each of these steps, a communication cost may be estimated using the communication model defined in Section 5.3, assuming that the elementary object is the elemental data level. This assumption is made since the remaining program has now been cast into the elemental data level, and because this seems the most appropriate level given that calculations are with respect to each element in the mesh.

6.6 Application of Parallel Strategy

Analysis of the data level movements that occur in the upwinding procedure show that IPC occurs due only to the interaction of the elemental and nodal data levels. Therefore we seek a way of projecting each aspect of the nodal data level into the elemental data level to reduce IPC, in the first three parts of the upwinding calculation.

However, all of this data has already been transformed into the elemental data level, from the communication reduction that had to take place for the parallelisation of the main body of the program. Therefore, no additional storage is required except for the calculation of intermediate variables.

6.7 Results and Discussion

Table 13 and Table 14 compare the communication costs before and after application of the parallelisation strategy. The calculation of the upwinding can be seen to be possible requiring only an increase in storage for the position of the centroids. There is no increase in IPC since the IPC noted in Table 14 is already required for the original algorithm. The utility of upwinding is demonstrated in Section 8.

Part which calculates	Movement	IPC/non-linear iteration
position of centroid	$N \rightarrow E$	> 0
centroidal velocity	$E, N \rightarrow E$	> 0
distance h	$E, N \rightarrow E$	> 0
Calculate Pe and α_{opt}	$E \rightarrow E$	0
Modify existing shape functions	$E \rightarrow E$	0

Table 13: Data level movements for upwinding. This is discussed in Section 6.7.

Part which calculates	Movement	IPC
Processed before upwinding:		
Restructure nodal co-ordinate data	$N \rightarrow E$	$3\sigma\gamma_e\kappa$
Restructure flow variable data	$N \rightarrow E$	$8\sigma\gamma_e\kappa$
Upwinding calculation:		
position of centroid	$E \rightarrow E$	0
centroidal velocity	$E \rightarrow E$	0
distance h	$E \rightarrow E$	0
Calculate Pe and α_{opt}	$E \rightarrow E$	0
Correct existing shape functions	$E \rightarrow E$	0

Table 14: Data level movements for final upwinding. This is discussed in Section 6.7.

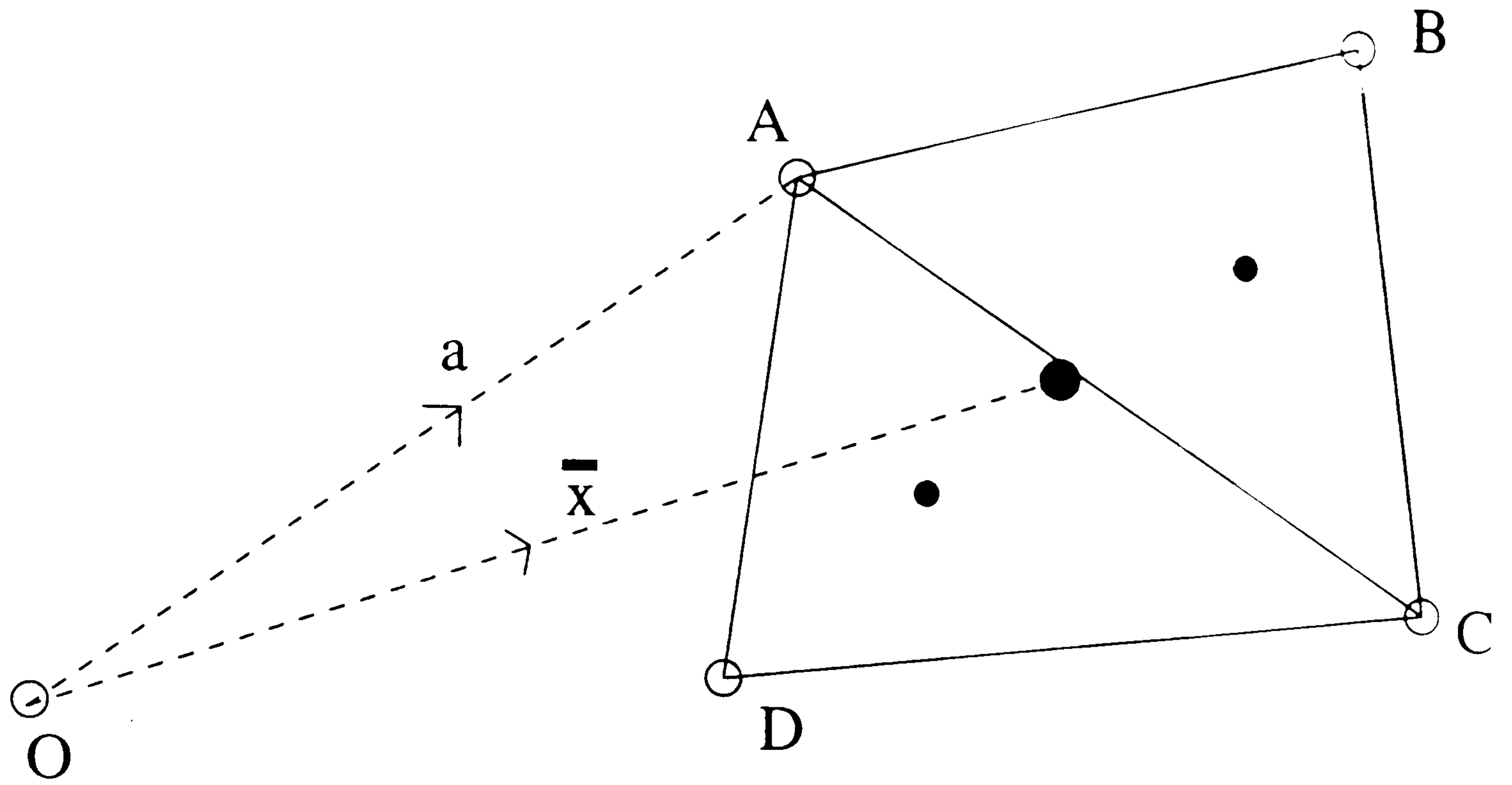


Figure 30: Calculation of the centroid of a quadrilateral element. This is discussed in Section 6.4.

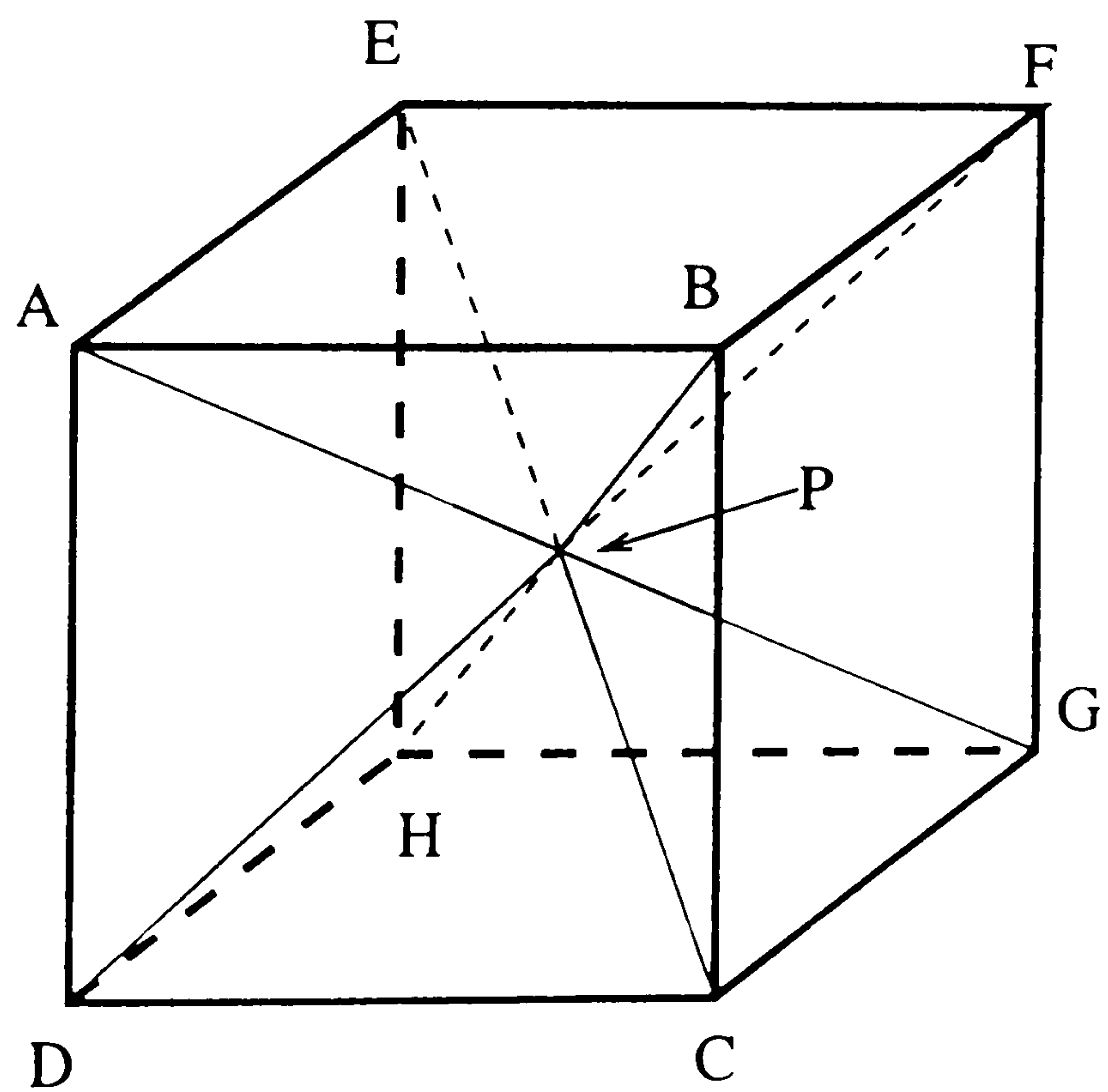


Figure 31: Calculation of the centroid of a hexahedral element. This is discussed in Section 6.4.

7 VERIFICATION OF THE PARALLEL CODE

7.1 Introduction

In this chapter we establish that the parallel code can produce numerically identical solutions to the sequential code depending on which iterative solvers are used. This is discussed in Section 7.2. In Sections 7.3 and 7.5, we go on to consider low Reynolds number backstep and cylinder flows. The results are then discussed in Section 7.6, including a comparison of the solution time for the parallel and sequential codes.

7.2 Comparison of sequential and parallel code

The optimised sequential code consists of a row-storage scheme with SOR, PCG and CGS iterative solvers available. The parallel code on the other hand has no storage scheme for the global stiffness matrix since no assembly of this matrix takes place. The same iterative solvers are available.

For the PCG and CGS iterative solvers, it should be noted that exactly the same numerical results are produced assuming that all other parameters are constant for an arbitrary problem. Though one might expect differences in values to be generated for catastrophic situations, for example, when the solution of the iterative solver may be diverging, this has not been observed. In order to verify this, three test cases were run. The meshes used are given below:

Case 1: 44 node, 10 element Couette-Poiseuille flow problem, as described in Section 2.11.1.

Case 2: 697 node, 640 element Poiseuille flow problem, as described in Section 3.4 (using mesh E).

Case 3: 1331 node, 1000 element three dimensional driven cavity problem, described in

Section 3.1.

The initial and boundary conditions are identical to those described in the relevant sections.

Case 1 is run using an implicit Crank-Nicholson scheme. The time step is 0.005 seconds, whilst the velocity and pressure relaxation is set to 0.8 and 0.4 respectively. For flow in a zero pressure gradient, a CGS-PCG method is used to solve the matrix equations resulting from the momentum and pressure correction equations. Some 10 iterations for both PCG and CGS. The problem is run for 80 time steps, with 5 non-linear iterations per time step. In case 2, the solution parameters are identical to those described in Section 3.4, with the exception of the solution of the linear systems of equations. Again, a CGS-PCG method is used, with 10 iterations for PCG and 10 iterations for CGS. In case 3, the flow is calculated at a Reynolds number of 1.0. The time step is 0.05 seconds, whilst the velocity and pressure relaxation parameters are set to 0.6 and 0.1 respectively. A CGS-PCG strategy is used to solve the linear systems of equations, with 10 iterations for CGS and 15 iterations for PCG. In all cases, mass lumping was used.

For a general flow variable ϕ , the average error (ϵ) between solutions generated by the sequential and parallel implementation may be estimated by

$$\epsilon = \sum_{i=1}^n \frac{\phi_{i,p} - \phi_{i,s}}{\phi_{i,s}}. \quad (89)$$

where n is the number of nodes, and $\phi_{i,p}$ and $\phi_{i,s}$ are the values of ϕ at node i produced by the parallel and sequential implementations respectively.

In each case, the average error for velocity and pressure was zero. This is not surprising since the PCG and CGS formulations are algorithmically exact on both sequential and SIMD architectures, and because double precision is always used on both architectures. This demonstrates the validity of the parallel implementation.

However, the SOR method is a flow-dependent algorithm, and as explained in Section 4.5.3 it cannot be directly implemented onto a SIMD platform without red/black colouring. Therefore the sequential and parallel implementations may produce different answers. In Figures 32 and 33 the residuals from the solution of matrix equations by both SOR (sequential) and red/black SOR (parallel) techniques is shown. The matrix equations result from the first time step of a 20x20 element two-dimensional driven cavity problem. As can be seen in Figure 32 there is very little discrepancy between the two methods for matrices produced from the momentum equations. In Figure 33 there is a larger discrepancy. This is probably because the matrices generated from the pressure correction equation are much less diagonally dominant than those produced from the momentum equations. This would reduce the stability of CGS, making solutions more susceptible to small difference in the implementation.

These differences seem too large to be simply the result of round-off errors, but considering that the formulations are so different it is surprising that such good agreement may be found.

7.3 Backstep flow

The solution to the flow behind a backward facing step is a standard test problem that has been addressed by numerous authors. These include Armaly *et al* [79] who considers flows between Reynolds numbers 70 and 8000, Gartling [80] who considers the boundary conditions at a Reynolds number of 800, and Gresho *et al* [81] who shows that the flow at a Reynolds number of 800 is stable. In this section, we consider low Reynolds number flows where one vortex is generated. It should be noted however, that at higher Reynolds numbers, a second vortex develops which is downstream of the first vortex and on the opposite wall (see Gartling [80]).

7.3.1 Flow domain and mesh considerations

The flow situation is shown in Figure 34, where is a region 7.5 units long, and of height 0.5 units, from the inlet to the step of height h . The expanded channel of height 1.0 units extends 30 units downstream of the step itself. A unit velocity profile is imposed at the inlet, with no-slip conditions imposed on the solid surfaces as shown in the figure. At the outlet, the pressure is set to zero.

For this problem a mesh has been built in the I-DEAS pre-processor, consisting of five mesh areas as shown in Figure 35. The mesh is biased towards the boundaries in order to capture the velocity gradients at the walls. The bias factors are shown on the figure next to dashed line arrows. In the area $0 < y < 0.5$ the horizontal lines of the mesh are centrally biased with a factor of 0.1, whilst in the region $-0.5 < y < 0$ the mesh is biased towards the lower wall using a factor of 10. Also, in the inlet the mesh is horizontally biased centrally by a factor of 0.1. This is to allow the flow to properly develop towards the entrance of the pipe and to reduce the aspect ratio between mesh areas 1 and 2. The experimental evidence in Section 3.4 suggests that the number of elements used in the streamwise direction of this flow is not as critical to accuracy as the number of elements in the spanwise direction. Therefore, this biasing should not detrimentally affect the flow solution coming into region 2.

The number of elements in each area is defined by the parameters e_i , where $i = 1$ to 5, shown in Figure 35. Here, the values are $e_1 = 20$, $e_2 = 75$, $e_3 = 25$, $e_4 = 8$, and $e_5 = 16$ giving a total of 2560 elements and 2705 nodes. The mesh is shown in Figure 36. At the beginning of the solution process it is assumed that the velocity field is $u = 1.0$.

7.3.2 Solution parameters

For low Reynolds numbers it is possible to run a steady-state solution by simply using a large time step. A solution at a Reynolds number of 2, based on channel width, may be

run with a time step of 1000 seconds. A combination of CGS with 50 iterations for the solution of the velocity-generated matrices and PCG with 100 iterations for the solution of the pressure-generated matrices is sufficient for the residuals of the linear equation solutions to be below $O(-9)$ for velocity and $O(-7)$ for pressure. A fully implicit scheme is used, with relaxation parameters 0.6 and 0.1 for the velocity and pressure respectively. Some 250 non-linear steps are enough to allow the pressure increments to drop by four orders of magnitude, as can be seen in Figure 37. A monitor is placed just after the step and in the expanded channel. Monitor values are shown in Figures 38 and 39. From this we can see that at the monitor, the u -values converged very quickly (within 25 non-linear steps) whilst the pressure converges within an even shorter interval. This illustrates an interesting feature of the steady-state approach to solving a problem; if the solution will converge, then it will converge fairly quickly.

7.3.3 Results

The calculated flow field at the inlet is shown in Figure 40. As the fluid moves along the entrance of the pipe, viscous shear forces have more of a spanwise effect. This acts to slow the fluid down. Since $\frac{\partial u}{\partial x}$ is negative, there is a spanwise component of velocity towards the centre of the pipe which conserves mass. Therefore, the calculated flow indicates that close to the entrance, there is a spanwise component of velocity towards the centre of the pipe. This is entirely reasonable for the boundary conditions applied to the flow. The pressure contours at the inlet shown in Figure 41 also suggest an explanation. The pressure at the singularities at the corners of the inlet are computed to be extremely high. This creates a computational pressure force from the region of high pressure towards low pressure regions, such as in the centre of the pipe. Note however, that within a small streamwise distance, the pressure gradient becomes constant within the pipe and the velocity profile becomes constant. This suggests that the singularities have only a localised effect.

As the boundary layer goes past the backward facing step, it is forced to separate. We

would expect an area of recirculation to form and this is clearly shown in Figure 42, again for a Reynolds number of 2. In addition, we would expect to see a boundary layer within the recirculation zone, both against the step and at the bottom of the flow domain. This is not visible due to the coarseness of the mesh.

The boundary layer reattaches to the pipe and the flow redevelops in the streamwise direction until a parabolic profile is achieved indicating a balance of the viscous shear forces due to the no-slip condition imposed at the walls, and the pressure force which acts to accelerate the flow. This can be seen in Figure 43 where no spanwise velocity is observed, and the streamwise velocity gradient is zero.

This makes an ideal solution from which to calculate higher Reynolds number flows. For this case the Peclet number is 0.5 and so for a Reynolds number of 10 it would be 5. Hence calculation of a flow at a Reynolds number of 10 is not possible without using a different mesh or using an alternate form of discretisation, that is, upwinding. However, we postpone consideration of this flow until Chapter 8.

7.4 Driven cavity flow

Driven cavity flow has already been considered in the present work to study the effects of increasing problem size on solution time (see Chapter 3). However, no flow solutions have so far been presented. Here we consider flow at a Reynolds number of 10, based on the width of the cavity.

The flow domain and boundary conditions are shown in Figure 44. As shown in this figure, no-slip conditions are imposed on three walls, with a slip wall condition of $u=1.0$ imposed on the $y = 1$ boundary. The dotted lines in the corners of the mesh indicate biasing towards the walls. This biasing is quadratic and maps x equally spaced points, where $0 \leq x \leq 1$ to $f(x)$ where,

$$f(x) = 2x^2, \quad 0 \leq x \leq 0.5, \quad (90)$$

$$f(x) = -2(x - 1)^2 + 1, \quad 0.5 < x \leq 1. \quad (91)$$

Initially a flow solution at a Reynolds number of 10 is generated on a 40x40 element mesh. Some 80 time steps of 0.05 seconds each, with relaxation values of 0.8 and 0.2 on the velocity and pressure are sufficient to generate a converged solution for the velocity, using 10 non-linear iterations per time step. Upwinding is not necessary to generate this solution. For the linear equation systems, it is necessary to use 25 iterations of the CGS solver and 50 iterations of the PCG solver for the velocity-generated and pressure-generated matrices respectively. From the initial condition of $u=v=0$ everywhere, the average pressure increment of the flow field drops two orders of magnitude, settling down to a constant value. This is to be expected because of the singularities at the corners of the walls.

The calculated velocity field is shown in Figure 45. A recirculation zone can be seen at each of the lower corners. The rest of the flow recirculates around a vortex centre which is positioned about one third of the way below the moving wall. It is difficult to detect evidence of a boundary layer next to the static walls, though the viscous effect of the moving wall is shown clearly by the almost linear u -velocity profile above the vortex centre. In Chapter 8, we consider this flow at a Reynolds number of 100.

7.5 Cylinder flow

We now consider the flow behind a cylinder at a Reynolds number of 5 (based on diameter). The mesh used is shown in Figure 46. This mesh consists of 10 mesh areas, eight of which are around the cylinder, as shown in Figure 47. The mesh is biased on the lines radiating out from the cylinder so that the elements at the edges of the mesh

are fifteen times larger than those close to the cylinder. This mesh has 5166 nodes and 2448 elements. A unit velocity profile is imposed at the inlet, with freestream velocities imposed on the horizontal boundaries of the mesh. No-slip conditions are imposed on the cylinder itself, whilst at the outlet, the pressure is set to zero.

Starting from an initial velocity field of unit horizontal velocity, the solution initially ran fully implicit for 10 time steps, with 10 non-linear iterations per time step. The time step was initially set at 0.005. A combination of the CGS-PCG technique was used with (25,50) iterations each. After this, the solution was run for a further 80 time steps with a time step of 0.05.

The pressure field is shown in Figure 48. The field is qualitatively and quantitatively symmetric about the diameter of the cylinder parallel to the inlet flow direction. Starting from the inlet, in line with the axis of the cylinder, the pressure field shows an increase towards the upstream side of the cylinder. As we move around the cylinder, this pressure drops until just before the top of the cylinder. Continuing around the cylinder, there is an increase in pressure until a point just downstream of the cylinder, shown in the figure as point A. Comparing this to the solution generated by Fornberg [82], good agreement is found. The solution also shows kinks in the pressure contours that occur where the mesh areas meet, for example at points B and C shown in the figure. Moving from left to right, a pressure drop is predicted along the lines corresponding to $y = x$ and $y = -x$. This is clearly a non-physical prediction probably caused by the change in element size.

The velocity field is shown in Figure 49. The field is again symmetric about the diameter of the cylinder parallel to the inlet flow direction. The fluid is shown to approach the cylinder and stagnate at the upstream point. Fluid above and below this stagnation point moves smoothly around the cylinder before converging at the downstream edge. However, given the no-slip condition that is applied on the cylinder itself, we would expect a boundary layer to be visible close to the surface of the cylinder. This cannot be deduced from this solution suggesting that the mesh is too coarse in this region.

Based on experimental evidence, Panton [83] reports that at a Reynolds number of 4 (based on cylinder diameter), vortices appear in the region downstream of the cylinder. This implies that the flow is no longer behaving like a Stokes flow. However, no vortices are evident from the flow solution shown in Figure 49 which does look like a Stokes flow. In fact, based on wake length values from experiments performed by Dennis and Chang [84], the mesh used in this experiment has only one node in the each of the vortices that would be generated for this flow. Therefore, it is not surprising that the velocity field does not show any vortices.

7.6 Discussion

The solutions presented so far have all been based on relatively small meshes at moderate Reynolds numbers. In order to generate solutions at higher Reynolds numbers, it is necessary to apply upwinding and use finer meshes. This is facilitated by using the parallel code since it runs significantly faster. For example, for the backstep problem with 2560 elements, each non-linear iteration on a Sparc Station 10 takes 33.4 seconds, whilst on the MasPar each step takes 25.2 seconds. Since the Sparc Station 10 operates at 13.6 MFlops, and each processor of the MasPar operates at 0.0354 MFlops, this represents a speedup of 509 and a parallel efficiency of 12.4%, using equations (68) and (69) from Section 4.2. Note that the parallelisability of the program, defined by equation (67) in Section 4.2 cannot be evaluated since it is not possible to run the entire program on just one processor of the MasPar.

However with only 2560 elements, the processor grid has not been fully utilised. In the next chapter, flows at moderate Reynolds numbers are considered. These require finer grids than the solutions presented in this chapter, and therefore more elements. As a result, all of the processors will be in use, giving improved performance.

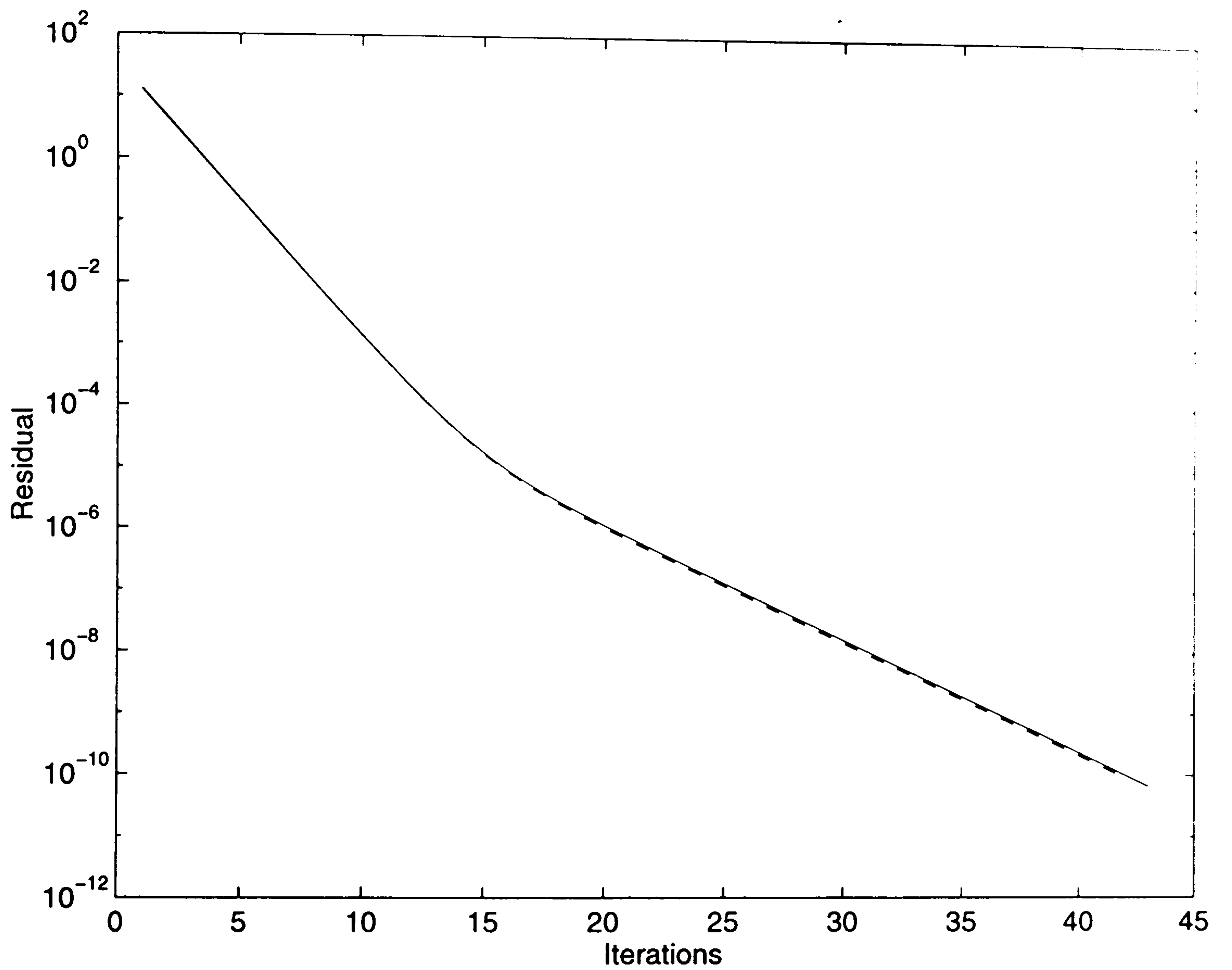


Figure 32: Comparison of residuals of the solution of the u-velocity system of linear equations from the sequential and parallel codes. The solid line shows residuals for the sequential implementation and the dashed line shows the residuals from the parallel implementation. This figure is referred to in Section 7.2.

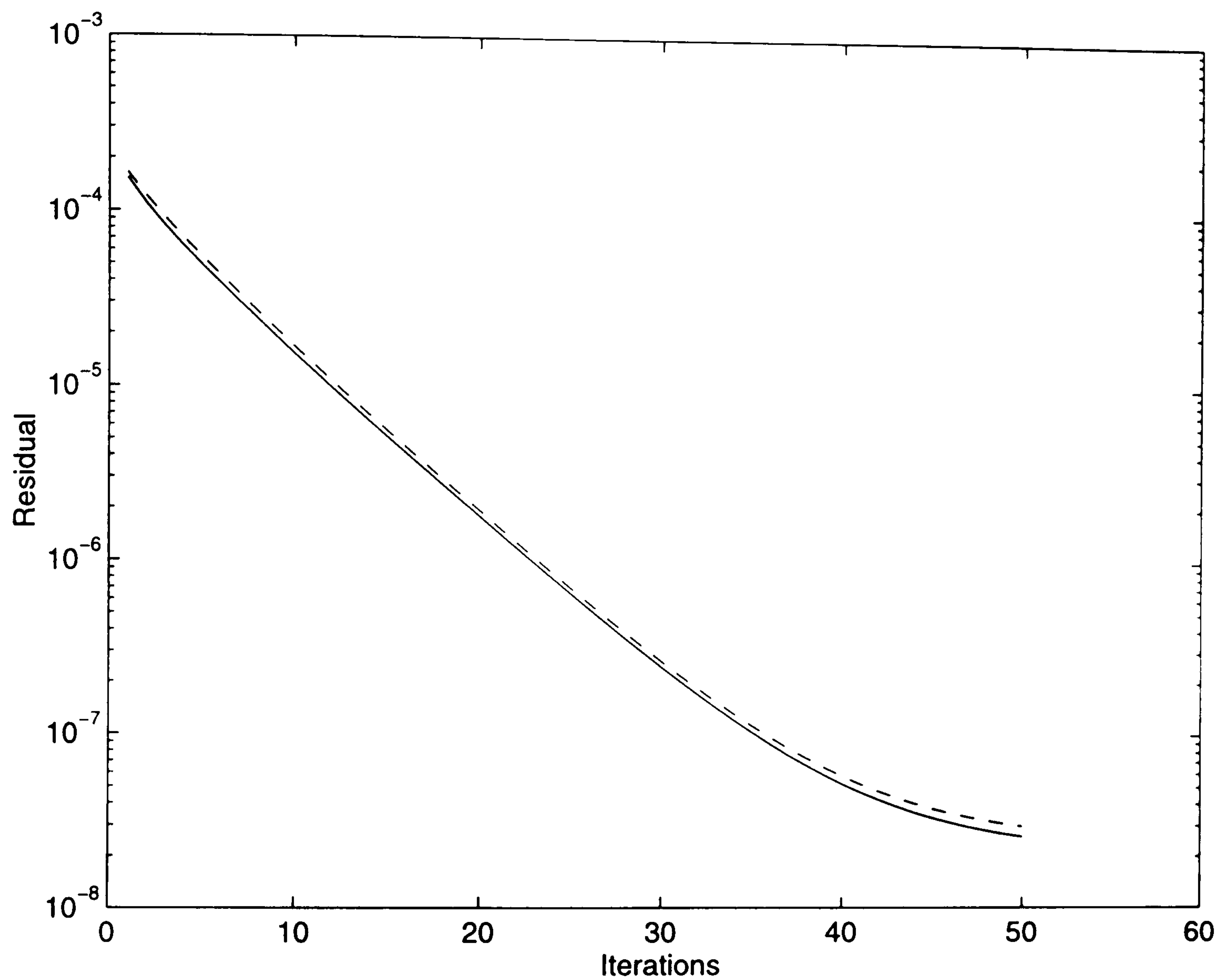


Figure 33: Comparison of residuals of the solution of the pressure-generated system of linear equations from the sequential and parallel codes. The solid line shows residuals for the sequential implementation and the dashed line shows the residuals from the parallel implementation. This figure is referred to in Section 7.2.

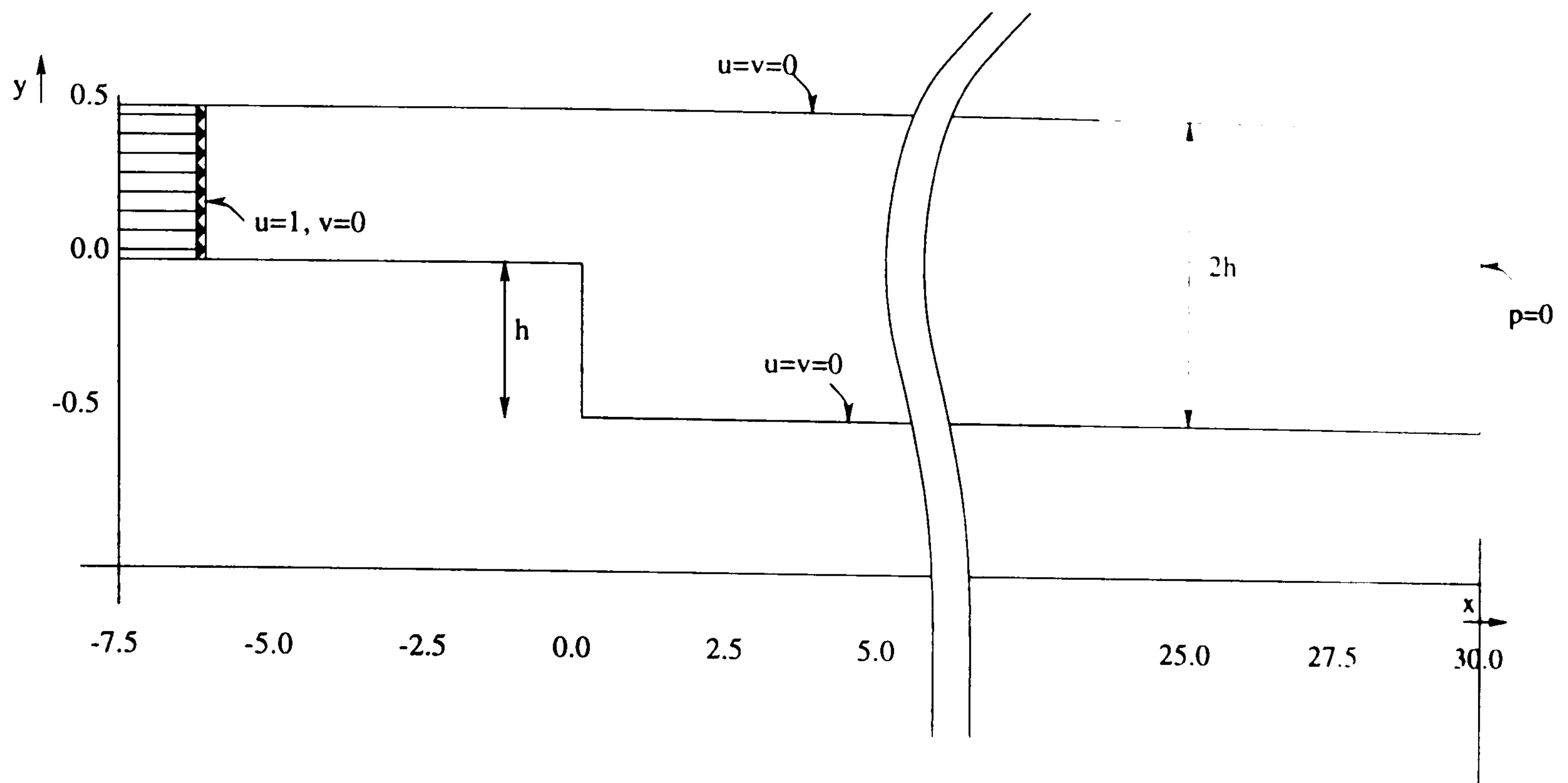


Figure 34: Backward-facing step geometry with boundary conditions. This is discussed in Section 7.3.1.

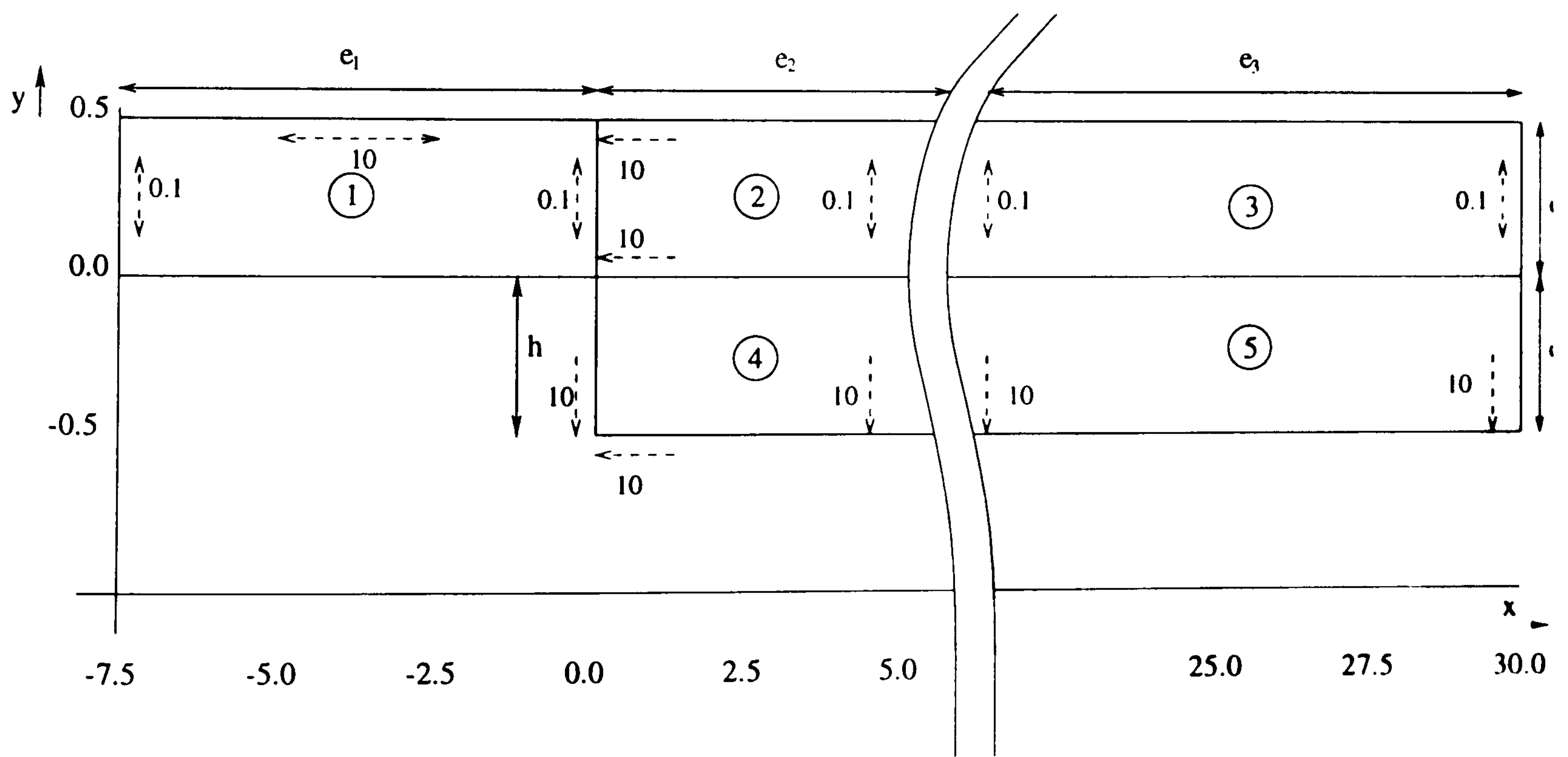


Figure 35: Backward-facing step mesh layout. The mesh consists of five mesh areas (shown by the circled number), with the number of elements along each edge shown by c_1 to c_5 . The dashed lines indicate the biasing used. This figure is referred to in Section 7.3.1.



Figure 36: Backward-facing step mesh. This mesh has 2560 elements and 2705 nodes.
This figure is discussed in Section 7.3.1.

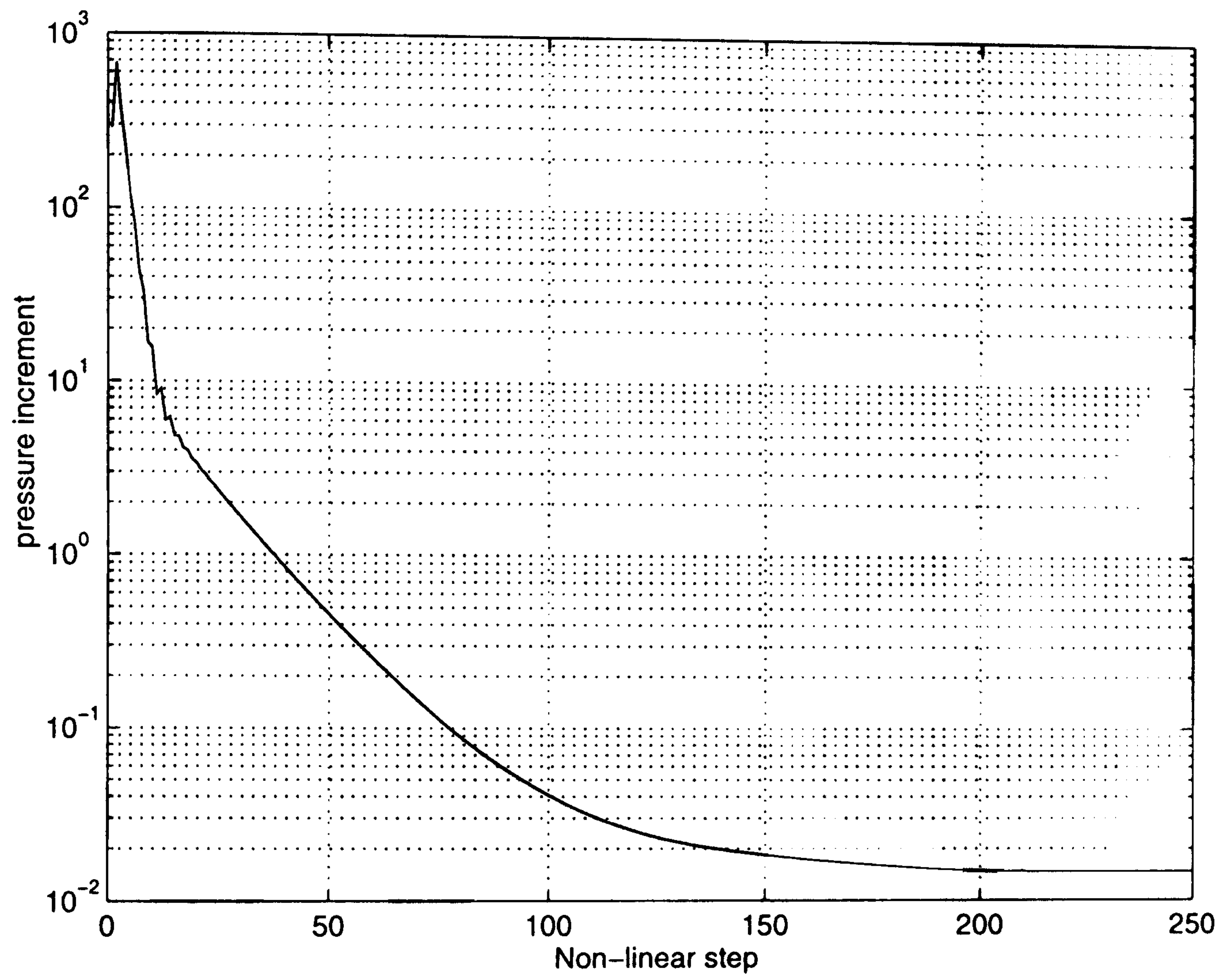


Figure 37: Pressure increments for backstep flow. This is discussed in Section 7.3.2.

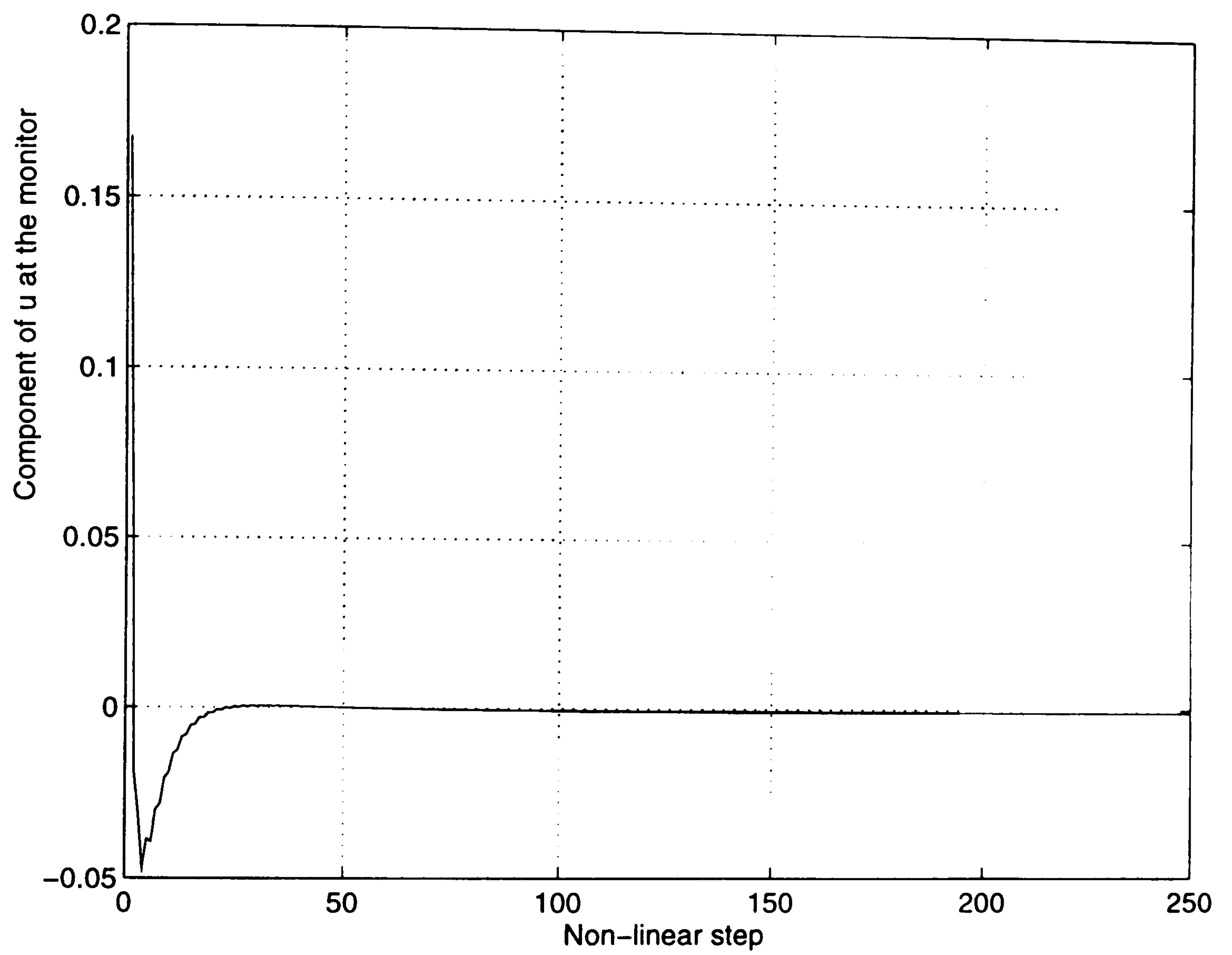


Figure 38: U-monitor values for backstep flow. This is discussed in Section 7.3.2.

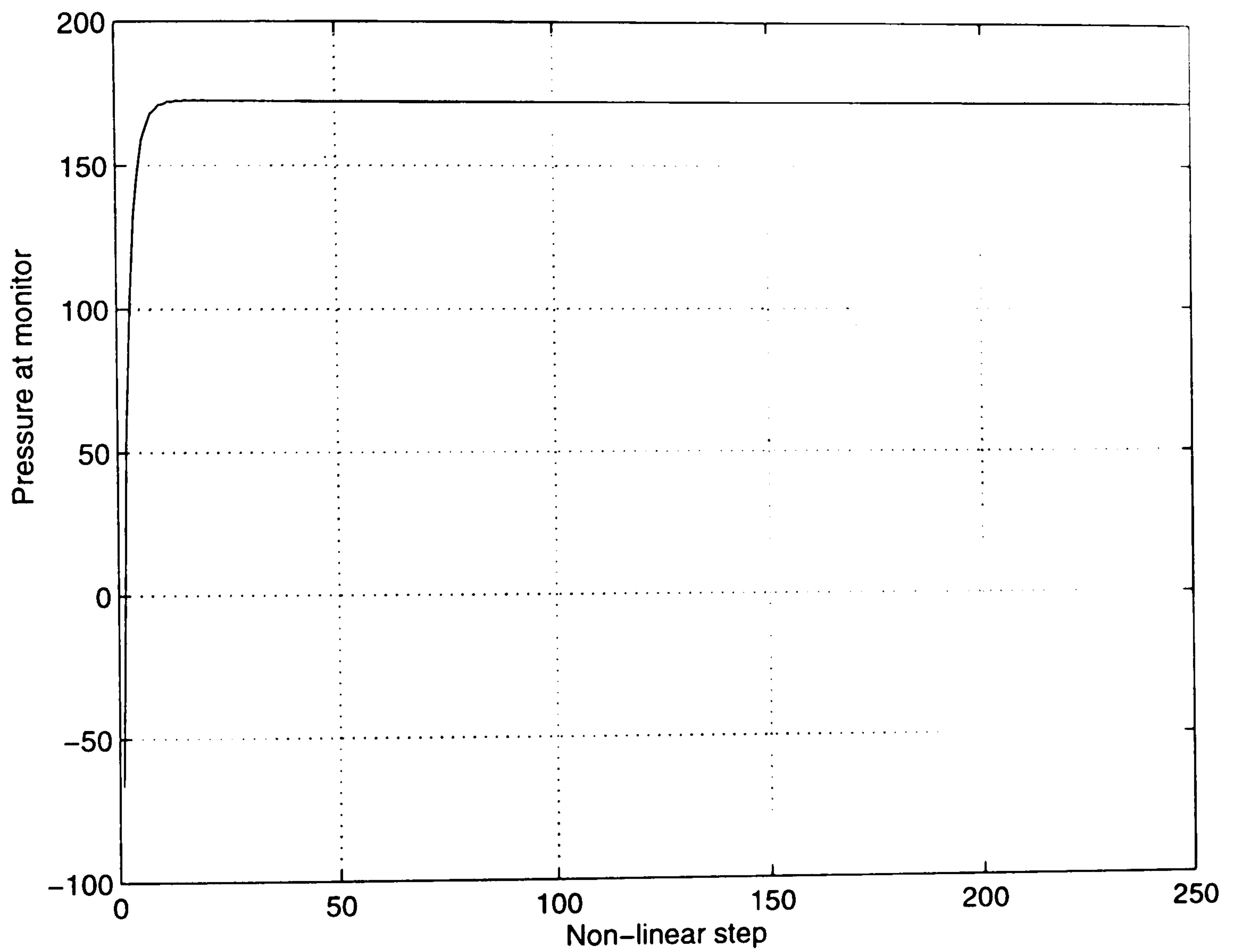


Figure 39: Pressure monitor values for backstep flow. This is discussed in Section 7.3.2.

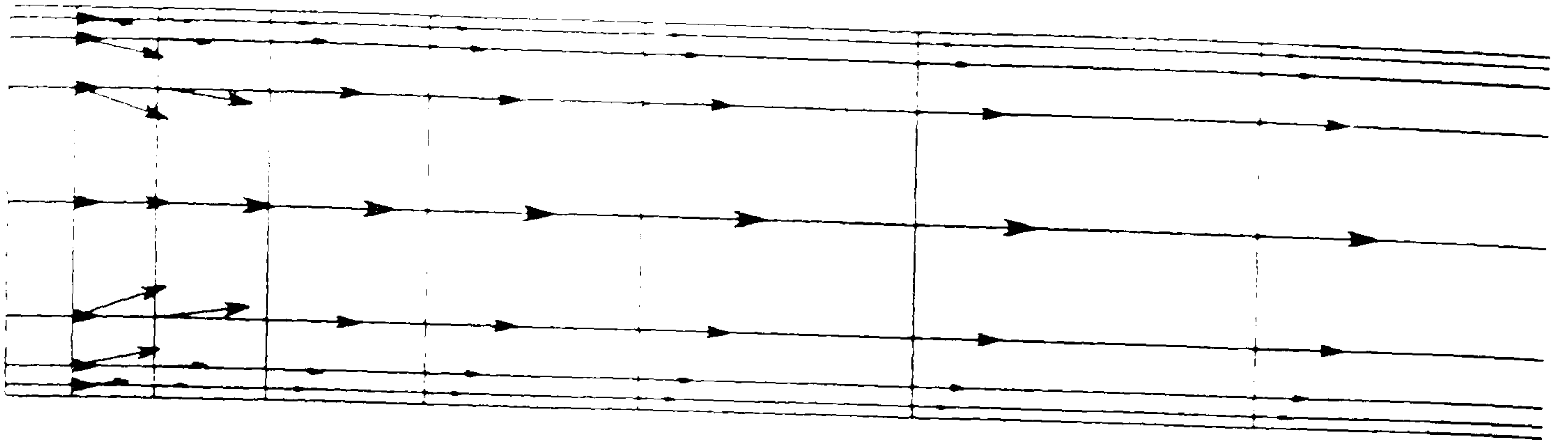


Figure 40: Velocity field at the inlet of a backstep flow. Scale 6 mm : 1m/s. This figure is discussed in Section 7.3.3.

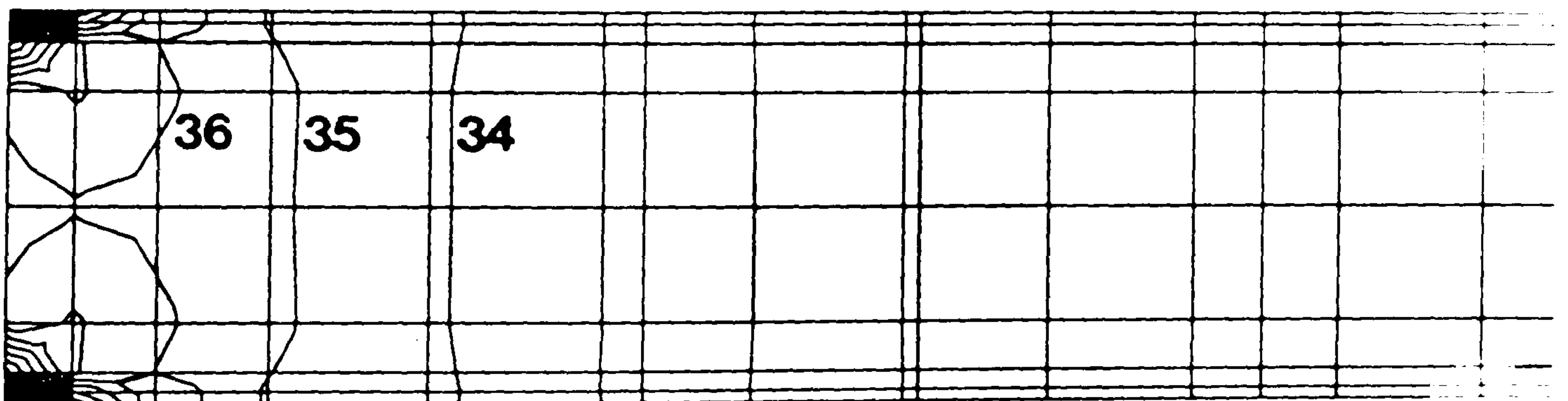


Figure 41: Pressure field at the inlet of a backstep flow. Some 50 contour levels were used, with values ranging from 15.25 to 762.5. Some examples of the pressure contour values, $p(\text{contour number})$ in the plot shown are: $p(36)=553.29$ and $p(33)=508.45$. This figure is discussed in Section 7.3.3.

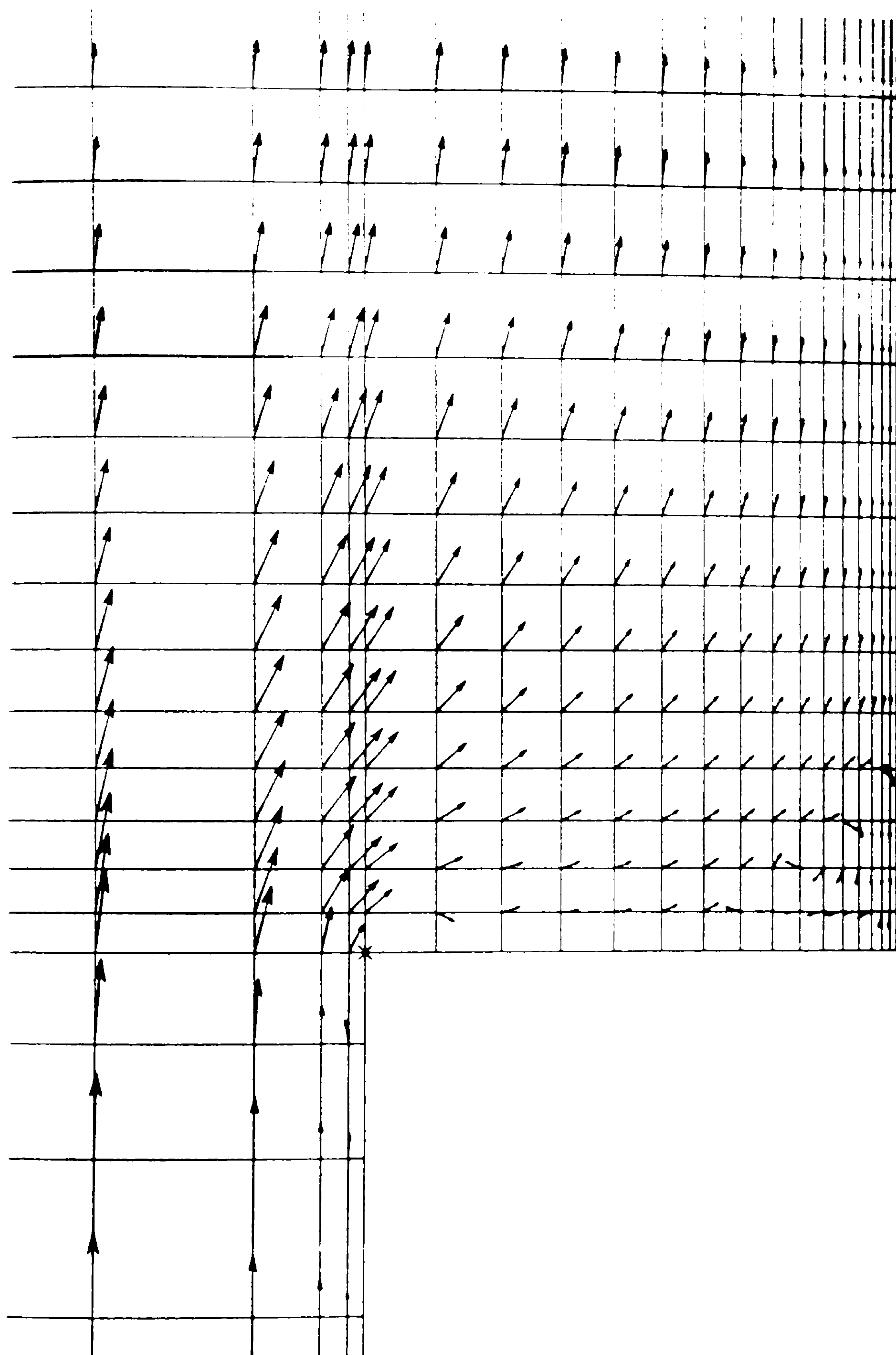


Figure 42: Close up view of velocity field near step in backstep flow. Scale 12 mm : 1 m/s. This figure is discussed in Section 7.3.3.

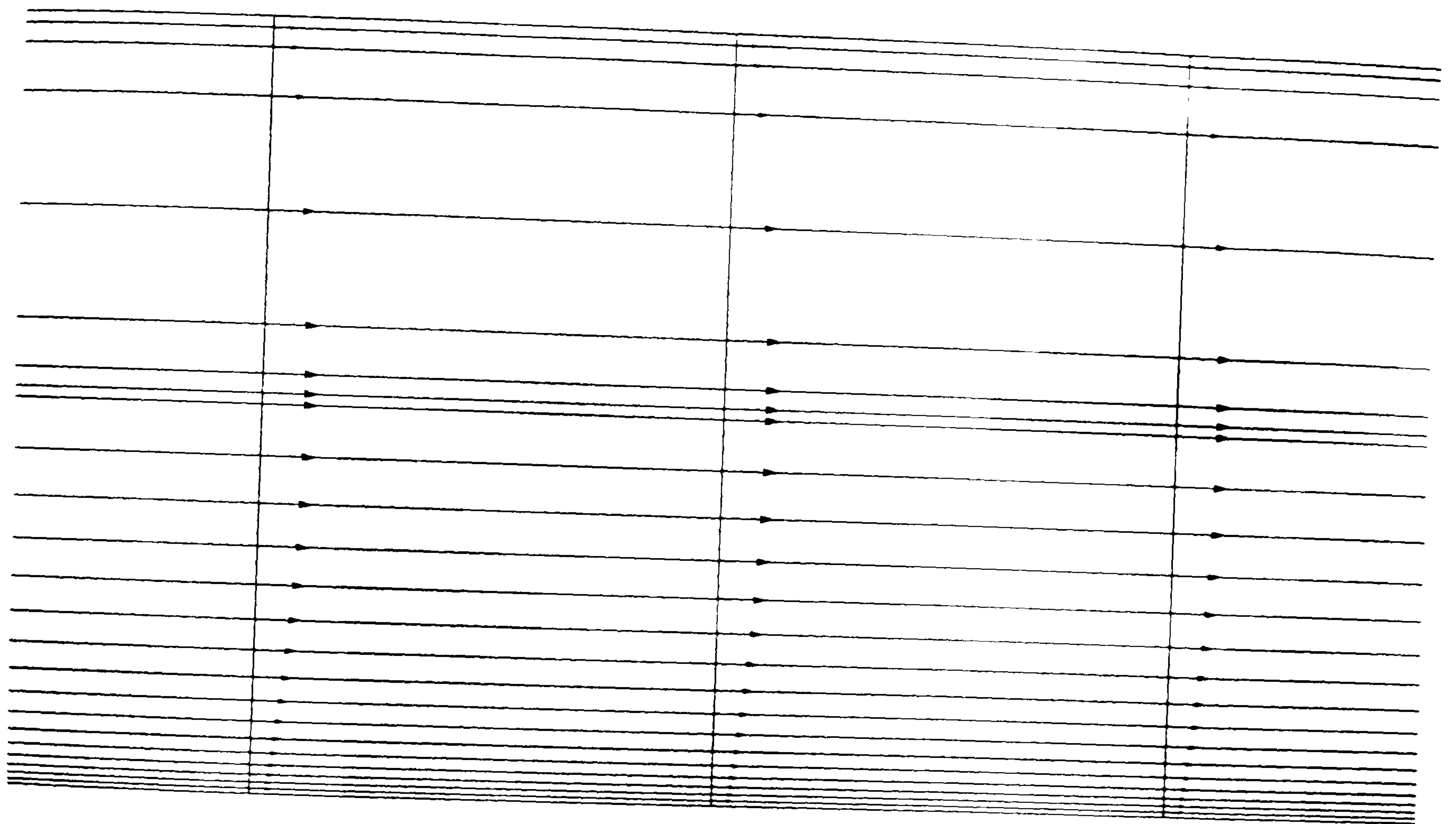


Figure 43: Velocity field at outlet for backstep flow. Scale 0.5 mm : 1 m/s. This figure is discussed in Section 7.3.3.

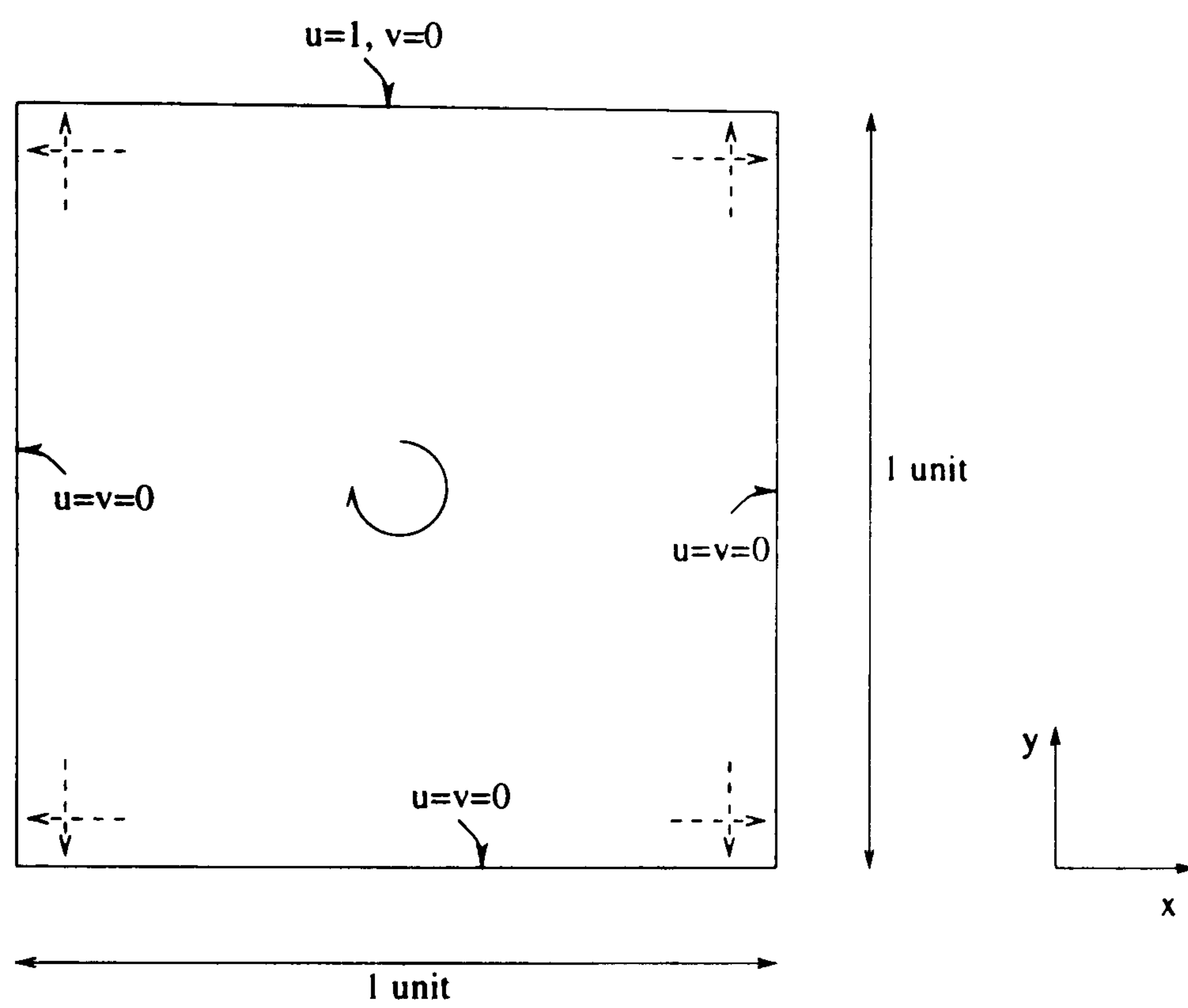


Figure 44: Domain and boundary conditions for 2D driven cavity problem. This figure is referred to in Section 7.4.

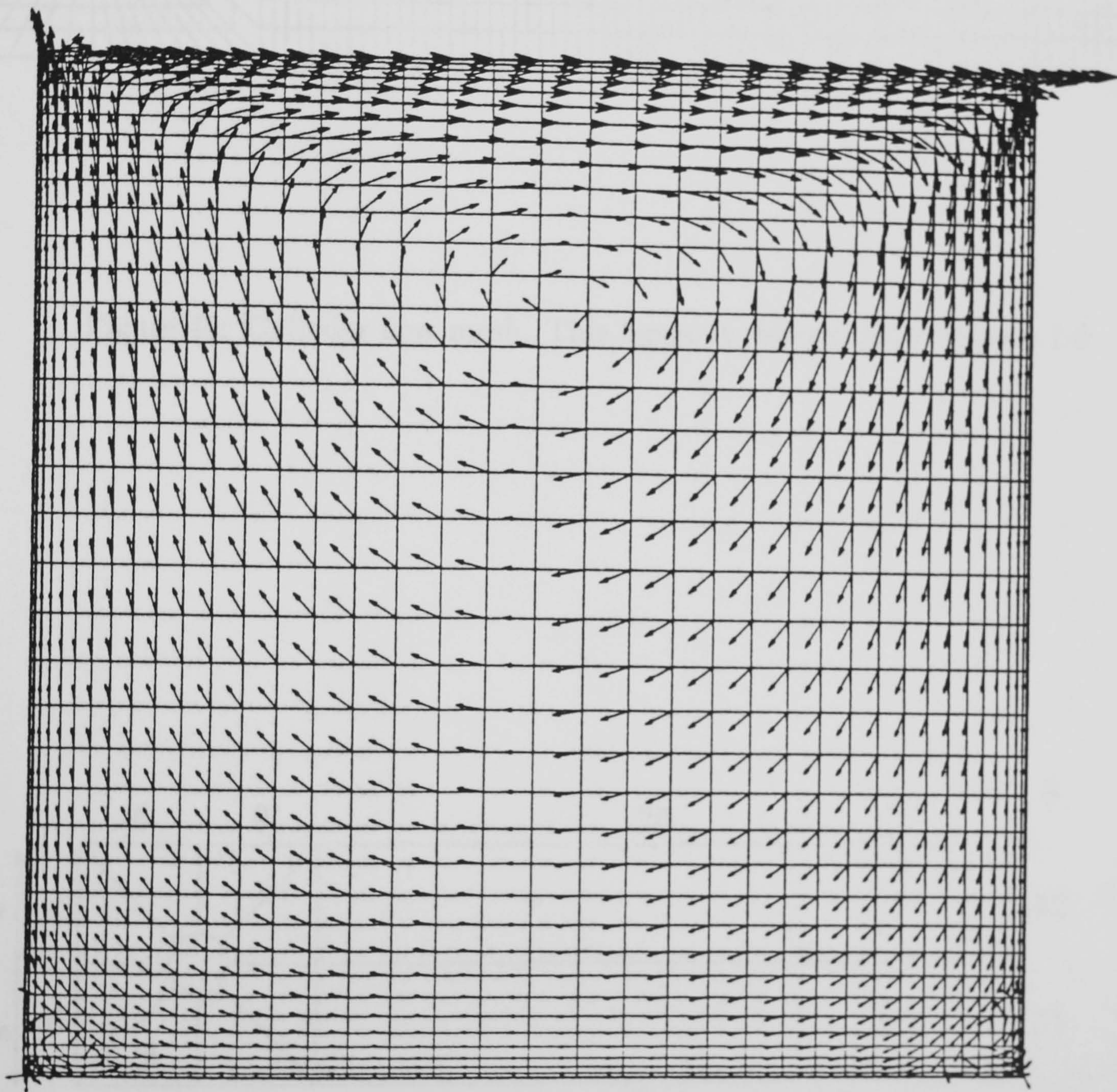


Figure 45: Driven cavity velocity field at $Re=10$. Scale 11 mm : 1 m/s. This figure is discussed in Section 7.4.

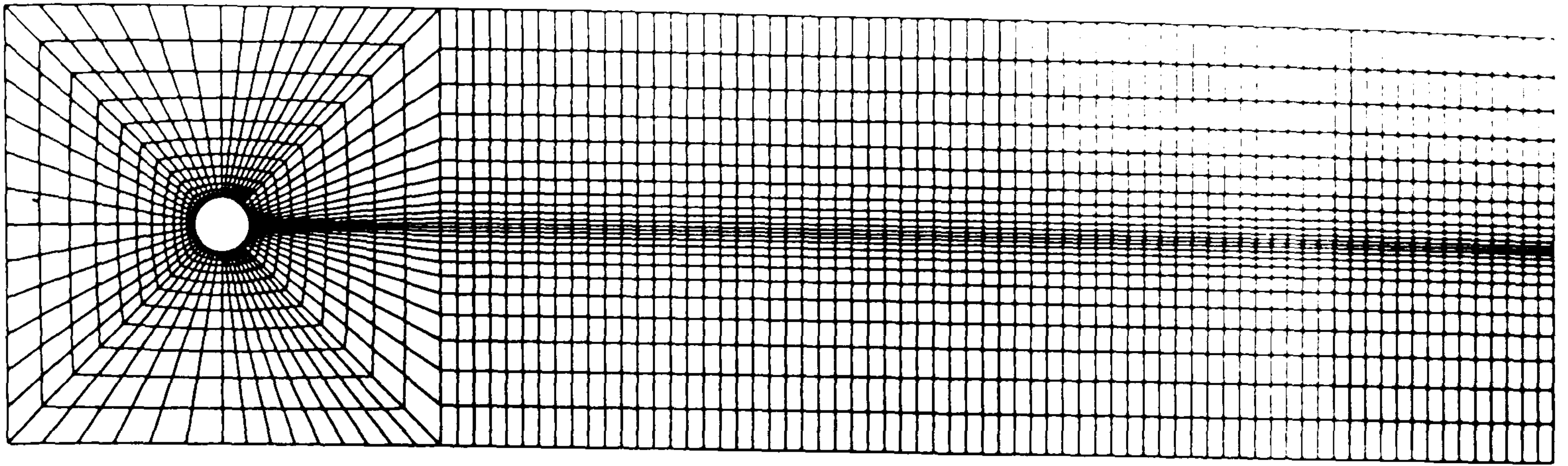


Figure 46: Cylinder flow mesh. This figure is referred to in Section 7.5.

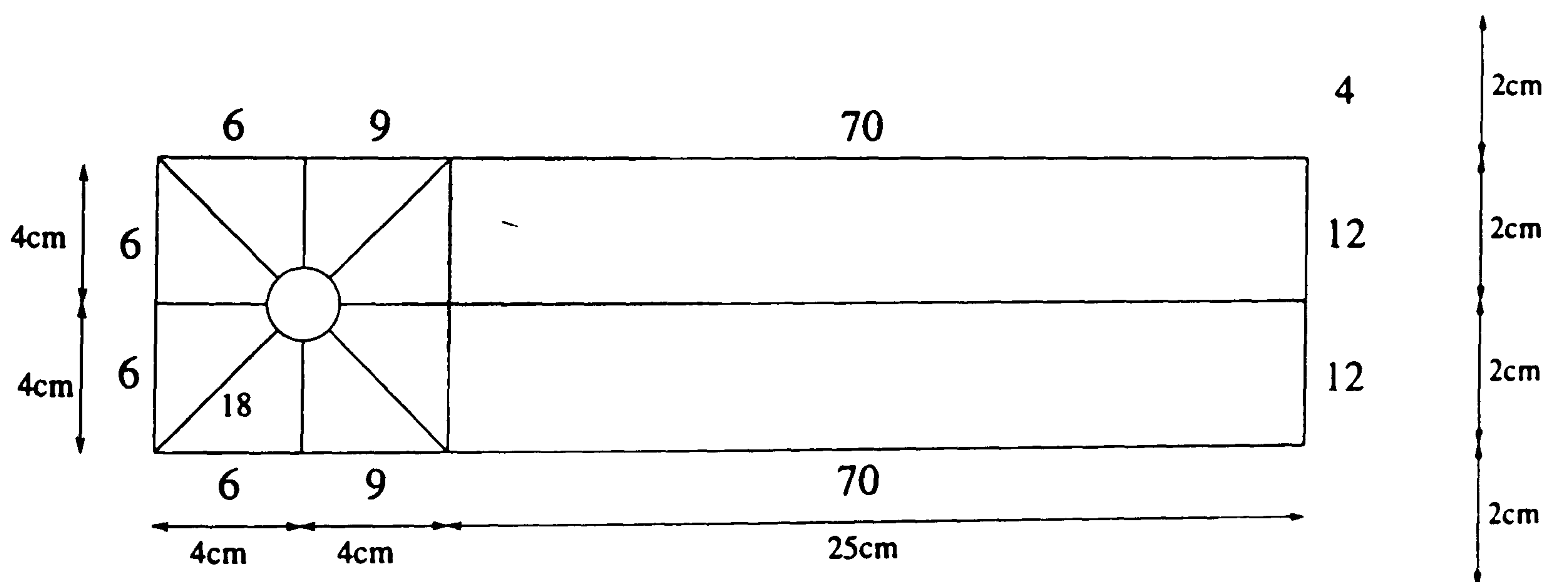


Figure 47: Cylinder flow mesh geometry. Not to scale. This figure is referred to in Section 7.5.

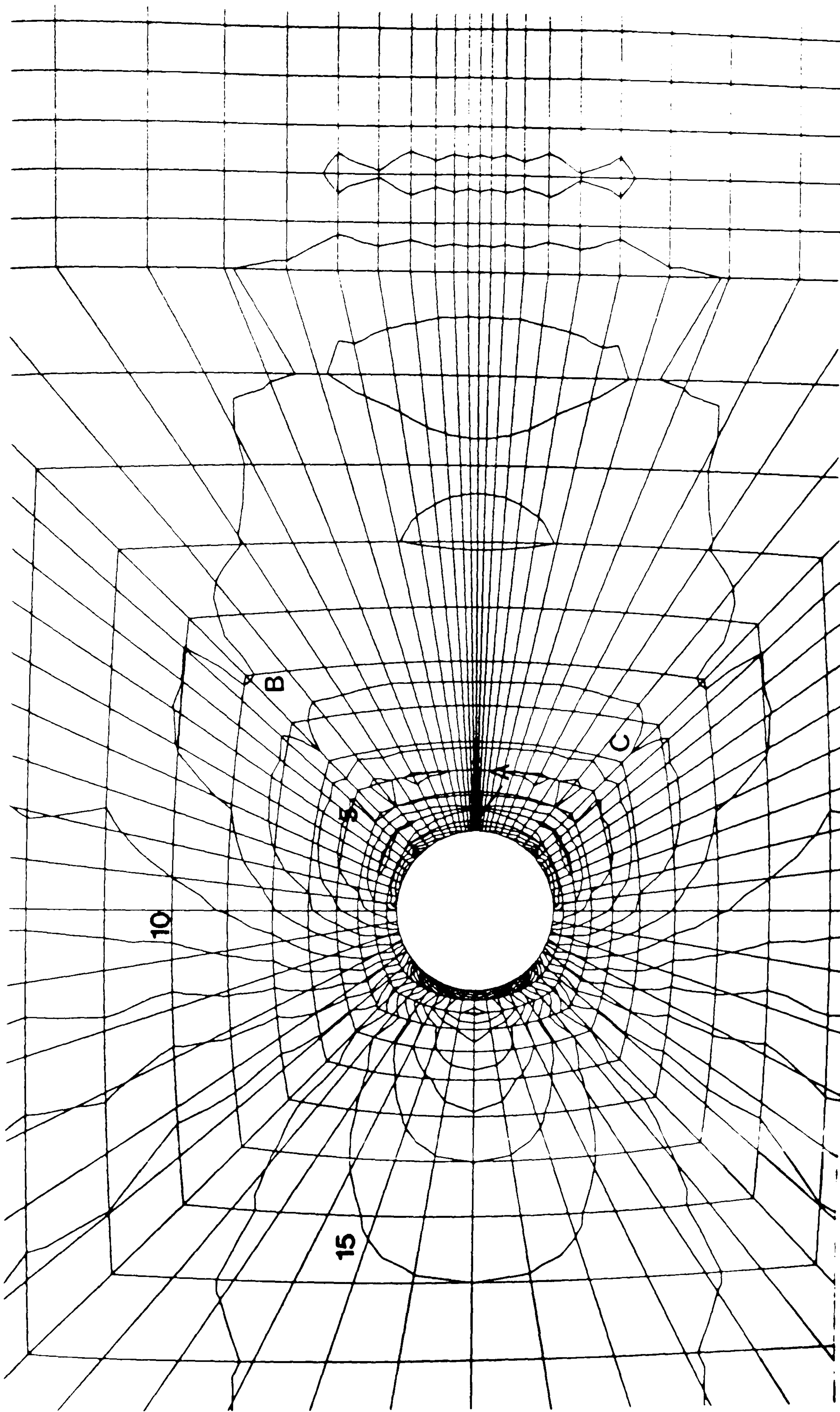


Figure 48: Pressure contours for cylinder flow at $Re=5$. Some 25 contours were used to generate this figure. Pressure values ranged from -2.26 to 1.73. Some values of pressure at contours, $p(\text{contour number})$ are: $p(5)=-1.462$, $p(10)=-0.6640$ and $p(15)=0.1340$. This figure is discussed in Section 7.5.

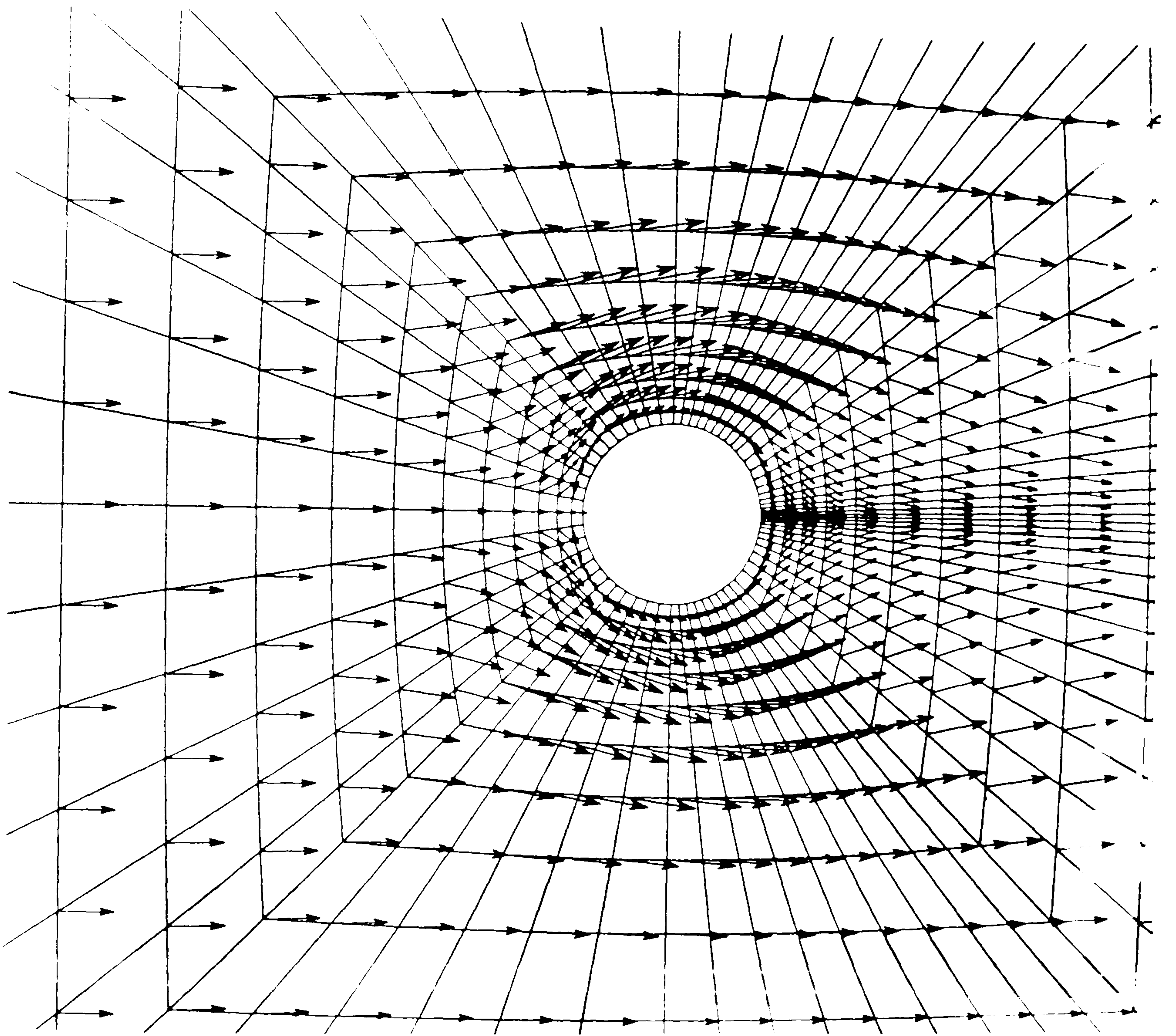


Figure 49: Velocity field for cylinder flow at $Re=5$. Scale 8 mm : 1m/s. This figure is discussed in Section 7.5.

8 MODERATE REYNOLDS NUMBER FLOWS

8.1 Introduction

In Chapter 7, the sequential and parallel versions of the codes were demonstrated to produce numerically identical solutions, unless SOR was used in the solution of linear equation systems. In this chapter, we will examine flows at higher Reynolds numbers. In order to do this, only the parallel code is used simply because it is faster than its sequential counterpart.

8.2 Driven cavity flow

Here we consider driven cavity flow at Reynolds number 100, based on the width of the cavity. The Reynolds number 10 solution presented in Section 7.4 is used as a starting point for this flow. The same mesh is used. Some 80 time steps with 10 non-linear iterations per time step are used. However, in order for this solution to converge it is necessary to drop the time step to 0.025, and drop the relaxation for velocity to 0.6.

We also seek to verify that solutions that are generated using upwinding are valid, and so upwinding has been used, although solutions to this flow have been generated without upwinding (see Shaw [19]). This practice can be improper. Use of upwinding implies that the convection terms are weighted in a manner which may not necessarily be valid for the required boundary conditions. However, in the formulation of Shaw [20], the boundary conditions are imposed after the upwinding, so this problem is avoided.

The resulting velocity field is shown in Figure 50. Comparing this to the Reynolds number 10 solution, it can be seen that the vortex centre has moved down and to the right. Moreover, the recirculation zone at the bottom right of the flow has grown slightly.

In order to verify the solution at this point, a comparison of the centreline u-values with values presented by Burggraf [85] indicates reasonable accuracy. This is shown in Figure 51.

This also demonstrates that the upwind weighting does not result in inaccurate answers.

8.3 Cylinder flow

The mesh used in Section 7.5 for this flow is quite coarse, and would be unsuitable for producing solutions at moderate Reynolds numbers. It appears that a large domain is needed to generate some characteristics of the cylinder flow solution. Nevertheless, some qualitatively correct flow characteristics have been obtained on this mesh and makes an ideal starting point for a progression with Reynolds number approach.

A unit radius cylinder is placed eight units downstream of the beginning of the flow domain, which extends 25 units downstream. The domain is 16 units wide. The geometry used is identical to the geometry suggested by Engelman and Jamnia [86]. Two meshes have been generated, using the I-DEAS pre-processor, and both meshes consist of twenty mesh areas. The number of elements along each of the sides defining the mesh areas is shown in Figures 52 and 53 corresponding to meshes A and B respectively. Thus mesh A has 1850 elements and 3888 nodes, whilst mesh B has 5548 elements and 11388 nodes. Biasing is applied on both meshes towards the cylinder, as shown in Figure 54. Both meshes are shown in Figures 55 and 56.

In Section 7.5, a solution at a Reynolds number of 5 was generated. This is projected onto mesh A. Where the domains of the meshes do not overlap, free stream velocity is assumed. Initially a flow solution at a Reynolds number of 10 is calculated. Some 500 time steps with 20 non-linear iterations per time step are used, with the time step set to 0.005. A combination of CGS-PCG technique was used with 50 iterations used for the solution of systems of linear equations generated by the momentum equations, and 200 iterations used for the solution of systems of linear equations generated by the pressure

correction equation.

When attempting to generate solutions at higher Reynolds numbers, the solution is projected onto mesh B. Using this as a starting point, however, flow solutions without upwinding rapidly diverge even with extremely small time steps. Since an increase in the Reynolds number suggests that convection is becoming more dominant in the flow, and since the Peclet number for this mesh is 3.4, it becomes necessary to apply upwinding.

8.4 Calculations with upwinding

Upwinding is necessary to restrict the formation and growth of wiggles that can appear in the solution because of a central-difference approach to solving problems where convection begins to dominate the flow. A clear demonstration of such problems is shown in Figure 57. This figure shows an attempt to generate a solution of the backstep problem at Reynolds number of 10, on a mesh where the Peclet number is 5.0. This is too high to permit a solution without upwinding. As can be seen, there is an oscillation which has formed at the top of the flow domain, beginning just after the step itself. This oscillation in the solution propagates downstream and is shown in this figure to die out. However, as the solution procedure continues, in time this wiggle corrupts the solution completely and leads to complete divergence. This may be prevented by using upwinding when the Peclet number for the flow becomes too high, as described in Chapter 6.

Reconsidering cylinder flow at Reynolds number 25, with upwinding used, the flow solution from Section 8.3 is used as a start for a flow at Reynolds number 25, using mesh B. Again 500 time steps with 20 non-linear iterations per time step are used, with the time step set to 0.005. The same parameters for the iterative solvers are used.

The resulting pressure field of the cylinder flow is shown in Figure 58. Many of the qualitative characteristics of the previously calculated flow are evident in this flow field. However, the benefits of refining the mesh are clearly visible in two ways. Firstly, the contours themselves appear to be much smoother. Also, though the kinks in the contours

which had been predicted are still evident where mesh areas join (for example at points **a** and **b**) these kinks are much less marked.

The velocity field is shown in Figure 59. Comparing this to the solution predicted at Reynolds number 5, we can see again, that many of the qualitative characteristics have been reproduced. However, several differences may be noted. The velocity field is definitely asymmetric about the spanwise diameter of the cylinder. This is largely due to the formation of two standing vortices downstream of the cylinder. Moreover, from the upstream stagnation point, the development of the boundary layer is clearly defined (for example, above the point **a** in the figure). As the boundary layer moves away from the upstream stagnation point, there is an acceleration due to the favourable pressure gradient around the surface of the cylinder. The velocity increases until a point before the top (or bottom) of the cylinder is reached, when the boundary layer begins to decelerate as the pressure recovers. This continues until the boundary layer no longer has sufficient momentum to stay on the surface of the cylinder and so it separates, leaving the two vortices. One of the separation points is marked **b**. The vortices themselves are generated by the resulting viscous shear force downstream of the cylinder. Within these regions of recirculation, we would expect to see boundary layers form along the side of the cylinder, and this is just visible (for example, at the point **c**).

The velocity field also suggests the following. The velocities of the two nodes above and below the upstream stagnation point (next to the point **d**) indicate that there is a flow towards the stagnation point, in a direction which has an upstream component. This kind of prediction has been apparent when generating solutions when a uniform flow meets two infinite parallel plates, using a grid that is too coarse in the streamwise direction. There is perhaps a direct analogy with the situation here where the elements adjacent to the upstream stagnation point are quite long and thin. Moreover, around the separation points the flow appears to be directed towards the cylinder. Both of these predictions would be eliminated with further mesh refinement in this region.

The length of the wake is a key parameter to cylinder flow. It is well-known that the

wake length is proportional to the Reynolds number up to about a Reynolds number of 40 (see Tritton [2]). In Figure 60 the length of the wake for the solutions at Reynolds number 10 and 25 are plotted along with both experimental and numerical data from Dennis and Chang [84]. The values obtained agree well with these published results.

8.5 Discussion

The parallel implementation has allowed investigation of flows at Reynolds numbers that could not be considered with the sequential code. For example, for the cylinder flow, each non-linear iteration on a Sparc Station 10 takes 193.7 seconds, whilst on the MasPar, each iteration takes 94.8 seconds. The parallelisability of the implementation, as defined by equation (67) cannot be determined since it is not possible to run the program on one processor of the MasPar. Nevertheless, using equations (68) and (69) from Section 4.2 this represents a speedup of 785.0 and a parallel efficiency of 19.2%, giving a reduction in processing time of over 50%.

However, despite the accuracy of the flows predicted for some moderate Reynolds numbers, attempts to produce solutions at higher Reynolds numbers failed. At Reynolds number greater than 40, the standing vortices behind the cylinder begin to oscillate and form a vortex street. This transient flow could not be produced. Repeatedly, the solution diverged even when using the comparatively good start solution from a Reynolds number of 25 and upwinding. This divergence is clearly shown in Figure 61 for a flow at a Reynolds number of 50. This figure shows the velocity and pressure at a point behind and below the cylinder, outside the recirculation zone. In the first five non-linear iterations, there appears to be very little change in the monitor values. The pressure then drops before rapidly increasing in value. The velocity then begins to increase leading to divergence. An analysis of element continuity indicates that once the flow begins to diverge, continuity is not being enforced.

There are several possible explanations for this. One of the problems associated with

cylinder flow at this Reynolds number is that the computational domain must terminate in a region of vorticity. In order to ensure the boundary conditions are as realistic as possible, it is no longer possible to maintain $p = 0$ across the outlet. Engelman and Jamnia [86] (who produced benchmark solutions for the cylinder problem at Reynolds number 100) suggest that equations (92) and (93) should be enforced at the outflow boundary.

$$-P + \mu \frac{\partial u}{\partial x} = 0, \quad (92)$$

$$\mu \frac{\partial v}{\partial x} = 0. \quad (93)$$

However, referring back to the formulation of Shaw's algorithm in Section 2.8, the value of the boundary terms in equation (32) is ignored. The advantage of this approach is that a natural Neumann boundary condition is automatically imposed unless a Dirichlet boundary condition is specified. The disadvantage is that boundary conditions such as equation (92) cannot be imposed without reprogramming to include the boundary terms of equation (32). This could lead to the undesirable scenario of reprogramming each time there is a change in boundary conditions. Alternatively, the discretisation of the Navier-Stokes equations may be performed in a different manner. For example, if the pressure gradient term is integrated by parts (as done by Gresho *et al* [52]) then equation (92) may be imposed directly. However, modifications of the original discretisation in this manner are beyond the scope of this work.

Instead then, attempts were made to generate flow solutions using the natural Neumann condition specified by:

$$\frac{\partial u}{\partial n} = 0 \quad (94)$$

where u is the velocity of the fluid and n is the normal to the boundary of the domain.

In addition, the pressure was zeroed at one point on the outflow boundary to set the value of the remaining pressures (thereby avoiding any potential singularity).

Using equation (94) automatically enforces equation (93) in a pointwise manner. Specifying zero pressure at one point of the outflow will locally satisfy equation (92), but will not necessarily enforce this on the rest of the outflow. Whilst this has not been a problem in generating solutions at some Reynolds numbers, convergence could not be attained for Reynolds numbers greater than 40.

Another possible explanation for the divergence of the solution is that the non-physical predictions manifest in the solution at Reynolds number 25 may be a source of error in the procedure at higher Reynolds number. It is further possible that these errors corrupt the solution leading to divergence, in analogy with the wiggles generated when upwinding is necessary but not used. This could be alleviated by further mesh refinement. However, this seems unlikely given the speed with which the solutions diverge as described at the beginning of this section.

One further possibility is the nature of the algorithm itself. Shaw's algorithm has been implemented using equal-order interpolation [19],[20]. In practical terms this represents a significant advantage over the more usual mixed formulations since only one mesh needs to be built, rather than a mesh for velocity and a mesh for pressure. Mesh generation may take a significant part of the total solution time (see Shaw [28]). Therefore this approach is desirable.

However, such schemes are regarded as violating the Babuska-Brezzi conditions which were discussed in Section 2.5.4. These two conditions are met only when unrealistic pressure modes are damped out of any generated solution, and when the number of unknown nodal values of velocity is greater than the number of unknown nodal values of pressure. This is the reason that many formulations for the solution of the Navier-Stokes equations are mixed.

Shaw reports that the first condition is satisfied by ensuring that the pressure correction

equation, equation (51) in Section 2.8. is only approximately solved and because the solution scheme is segregated [20].

However, consider equation (51). This may be rewritten as

$$(B^T L^{-1} B + E^T L^{-1} E) p_j'^{n+1} = \delta \quad (95)$$

where δ is the right hand side of equation (51), and where $L = \text{diag}(A)$.

Then if the entries of L can be negative then the left hand side of equation (95) may not be positive definite. Since $L = \text{diag}(A)$, consider equation (34). If the values of \bar{u} or v are zero or negative then L may not be entirely positive. This may occur for example in areas of recirculation or at stagnation points. Both of these situations occur in the cylinder flow at moderate Reynolds numbers. It is difficult to judge how likely it is for such a situation to lead to a divergence in the solution. An eigenvalue analysis of the matrices generated by the pressure correction equation would show if the left hand side of equation (95) ceases to be positive definite but this is beyond the scope of this work.

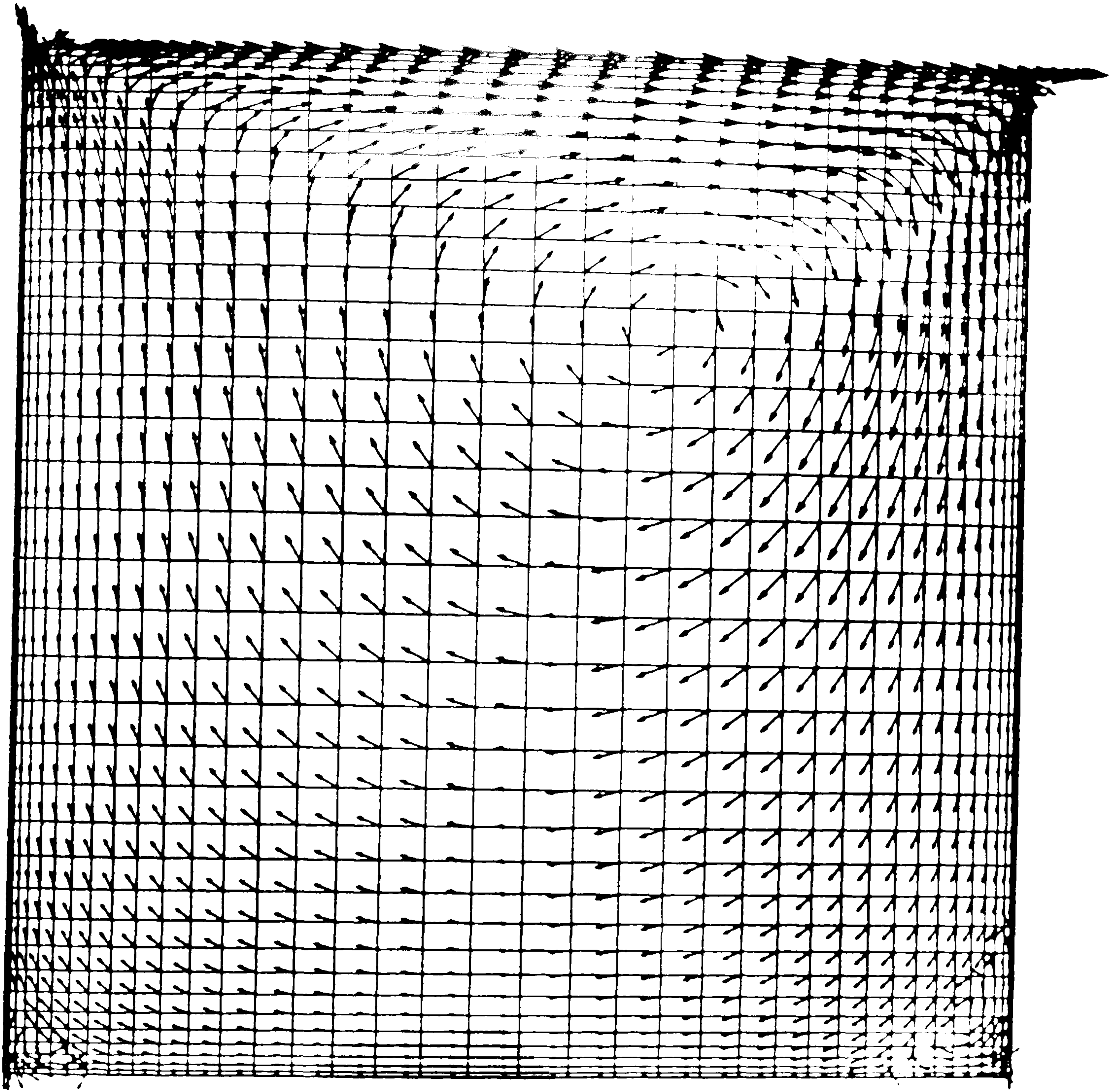


Figure 50: Velocity field for driven cavity flow at $Re=100$. Scale 11 mm : 1 m/s. This figure is discussed in Section 8.2.

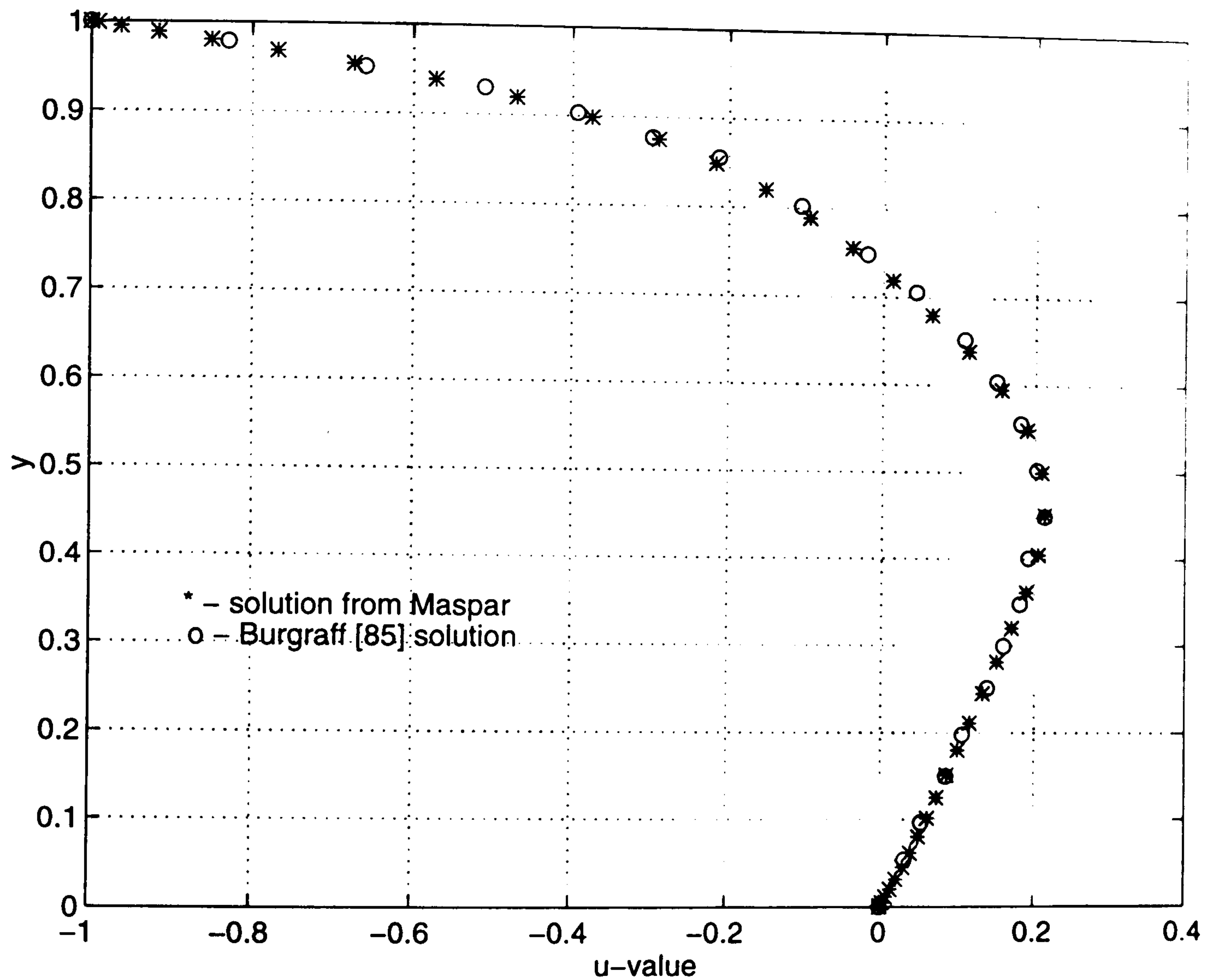


Figure 51: Centreline u -velocities for driven cavity flow at $Re=100$ compared with results from Burggraf [85]. Note that the calculated values for the velocities have been reflected in the line $u = 0$ to coincide with published results. This figure is discussed in Section 8.2.

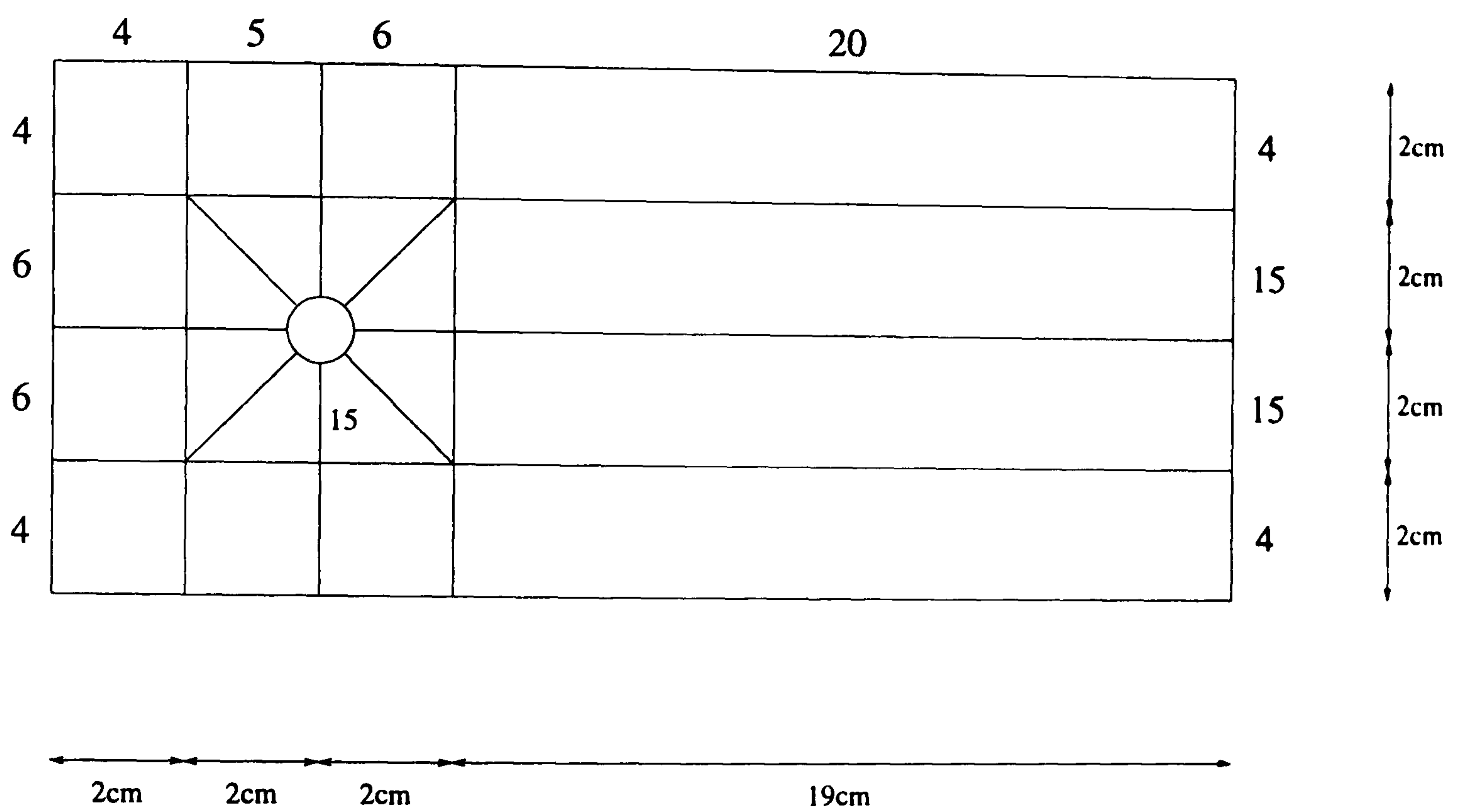


Figure 52: Cylinder flow mesh geometry. Units are shown in centimetres at the bottom of the diagram, and the number of elements along each side is shown by the dimensionless numbers. Not to scale. This figure is discussed in Section 8.3.

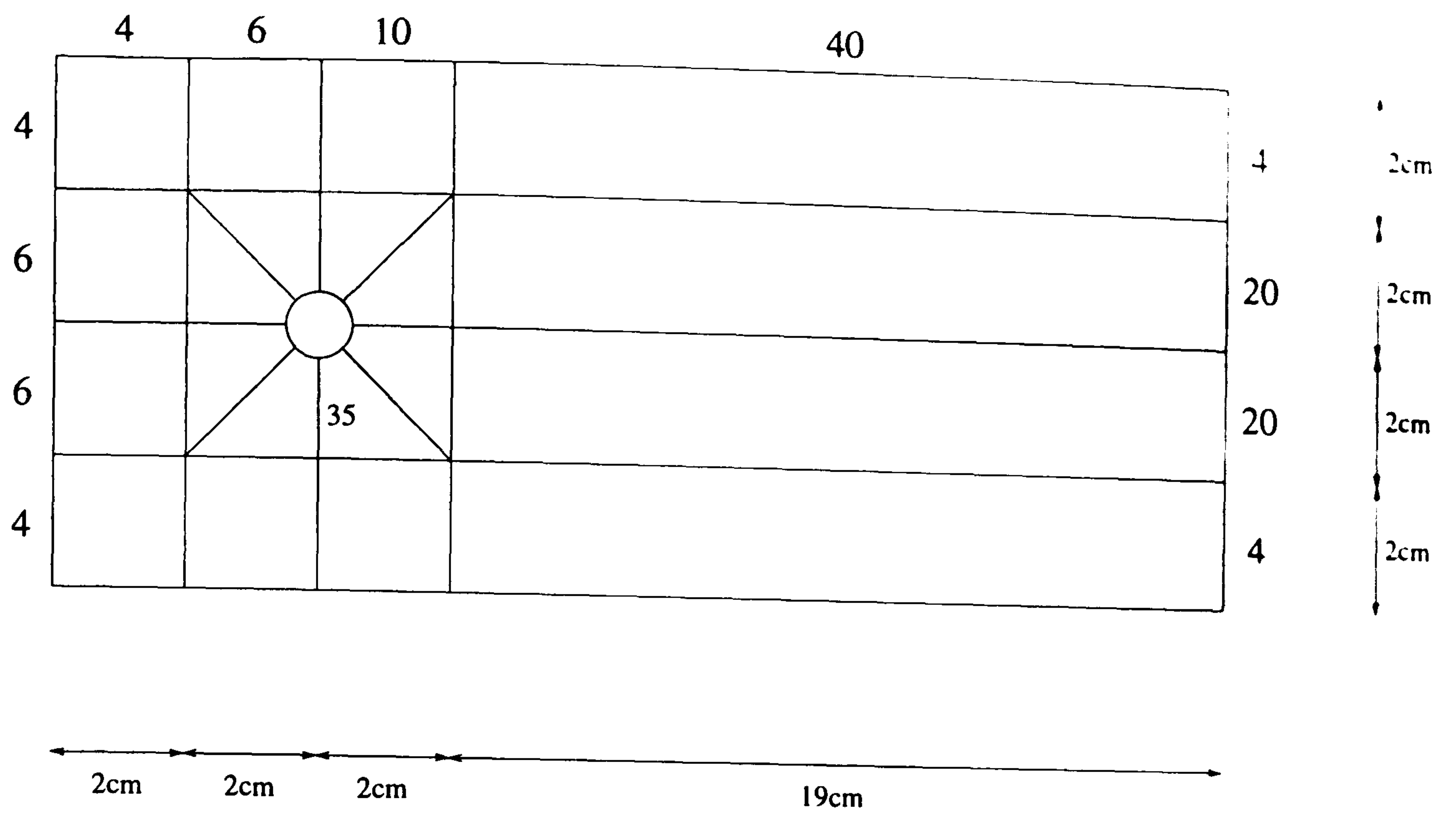


Figure 53: Fine cylinder mesh layout. This is referred to in Section 8.3.

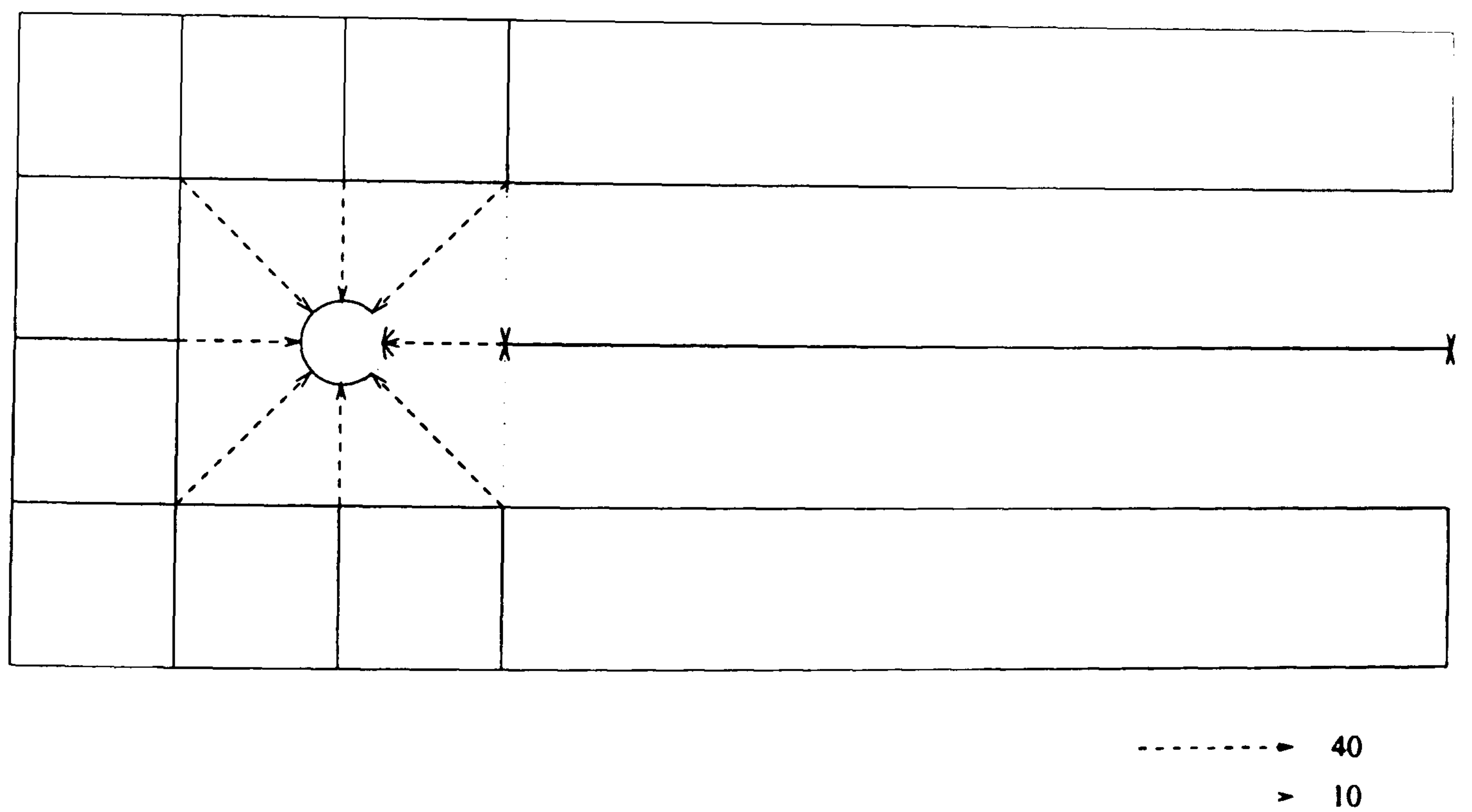


Figure 54: Cylinder flow mesh biasing. Only the dotted and dashed lines are biased. The biasing is in the direction of the arrow. This is referred to in Section 8.3.

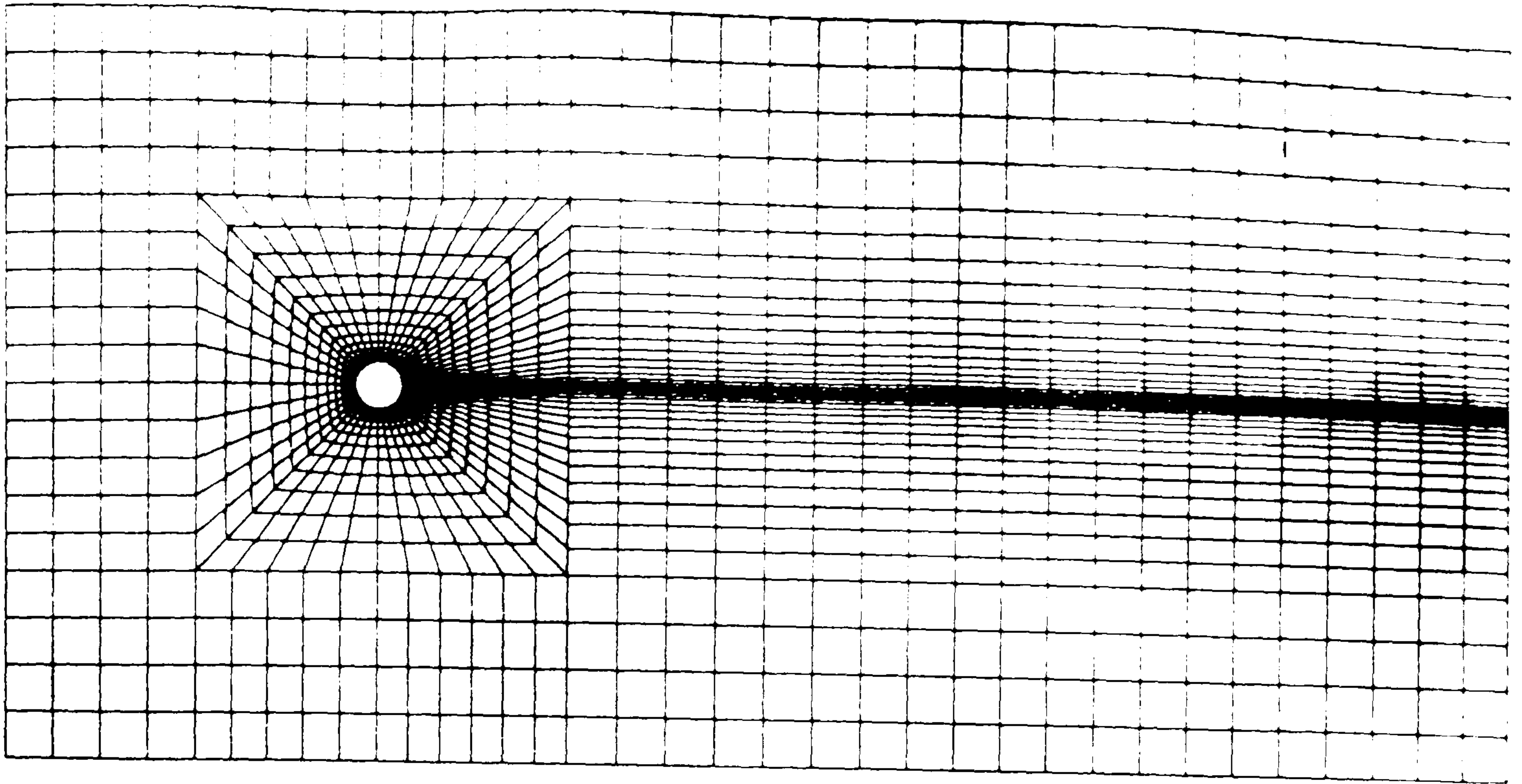


Figure 55: Coarse cylinder mesh. There are 1850 elements and 3888 nodes. This is discussed in Section 8.3.

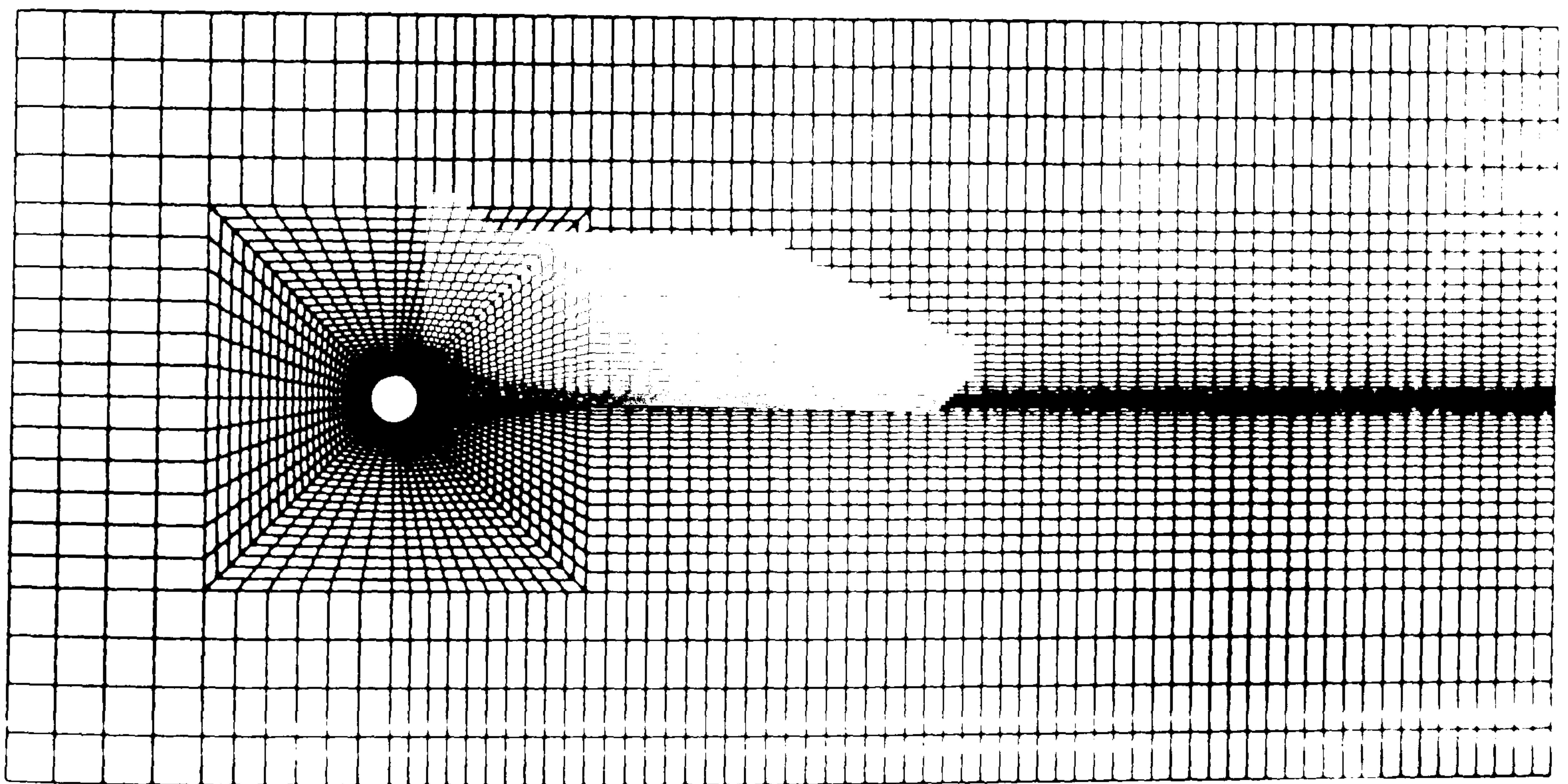


Figure 56: Fine cylinder mesh. There are 5548 elements and 11388 nodes. This is referred to in Section 8.3.

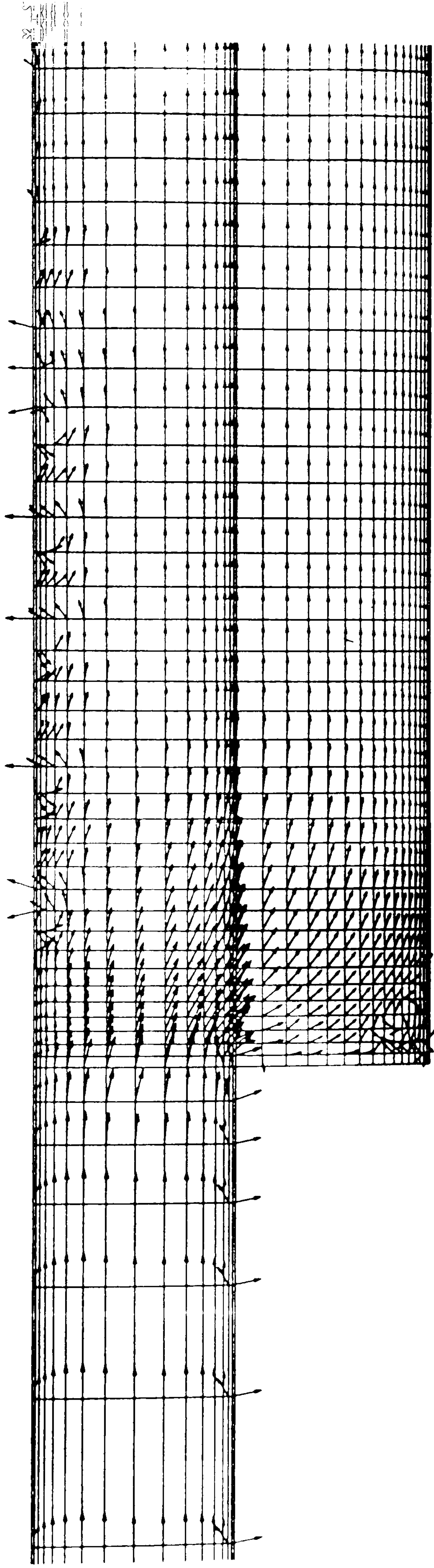


Figure 57: Wiggles in solution when the convection term is discretised with central differencing. The scale used is 4 mm : 1 m/s. This figure is discussed in Section 8.4.

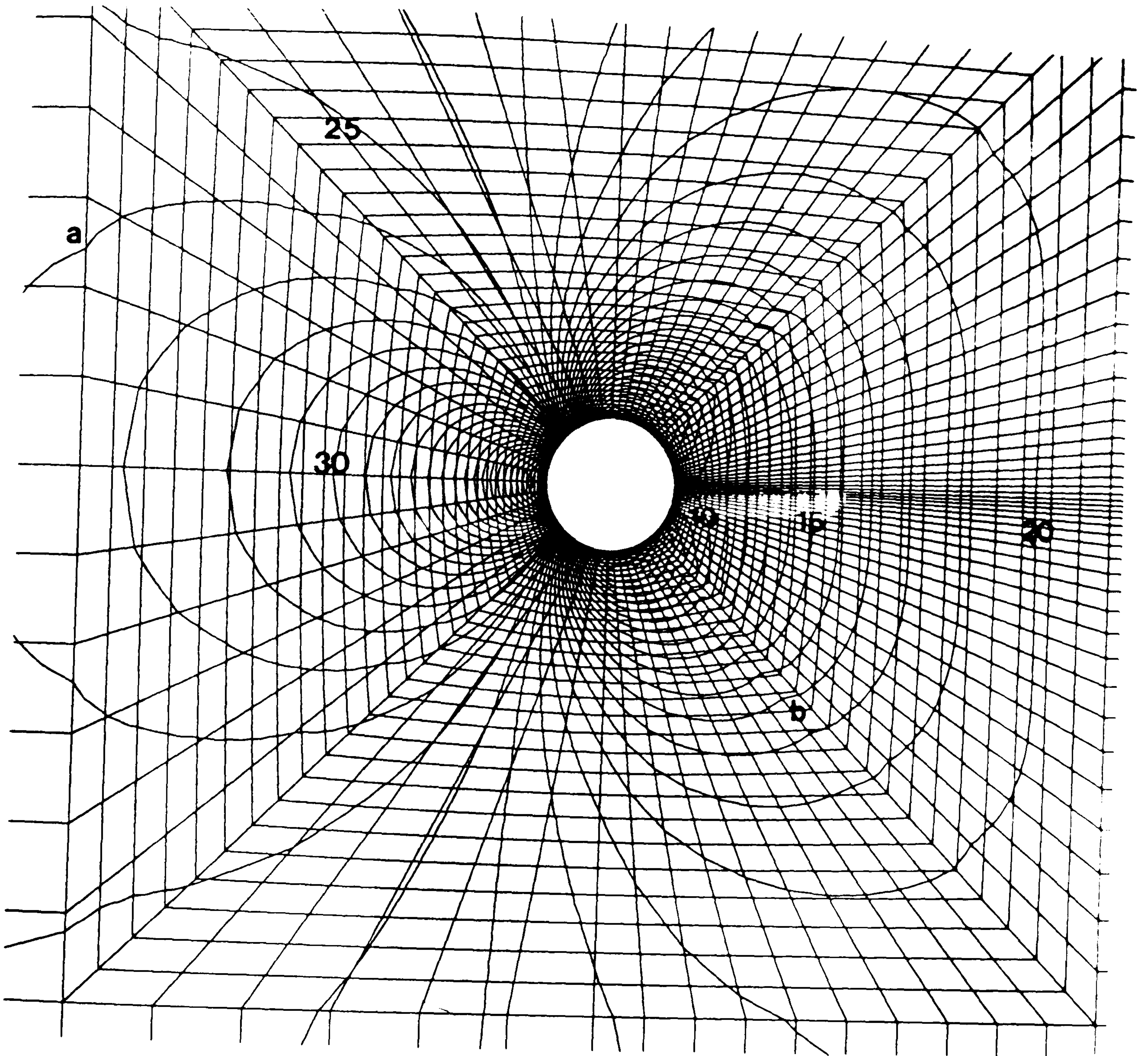


Figure 58: Pressure field for cylinder flow at $Re=25$. Some 50 contours were used to generate the contours. Only a few are labelled. Values of pressure at contours, $p(\text{contour number})$ are: $p(10)=-0.5426$, $p(15)=-0.3165$, $p(20)=-0.0904$, $p(25)=0.1358$, $p(30)=0.5880$. Note that the pressure boundary condition is $p = 0$ at the outlet. This figure is discussed in Section 8.4.

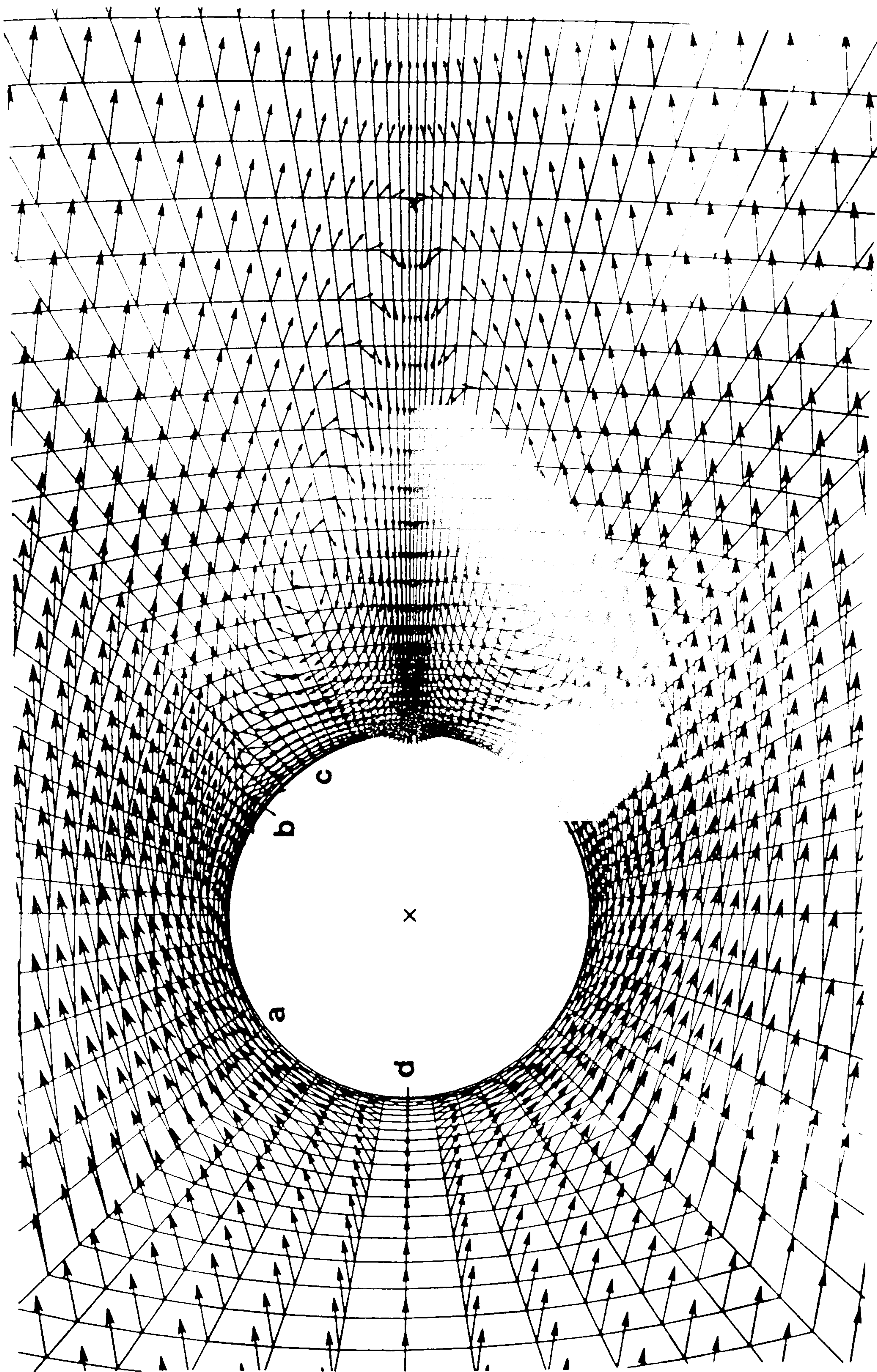


Figure 59: Velocity field for cylinder flow at $Re=25$. Scale 8 mm : 1 m/s. This figure is discussed in Section 8.4.

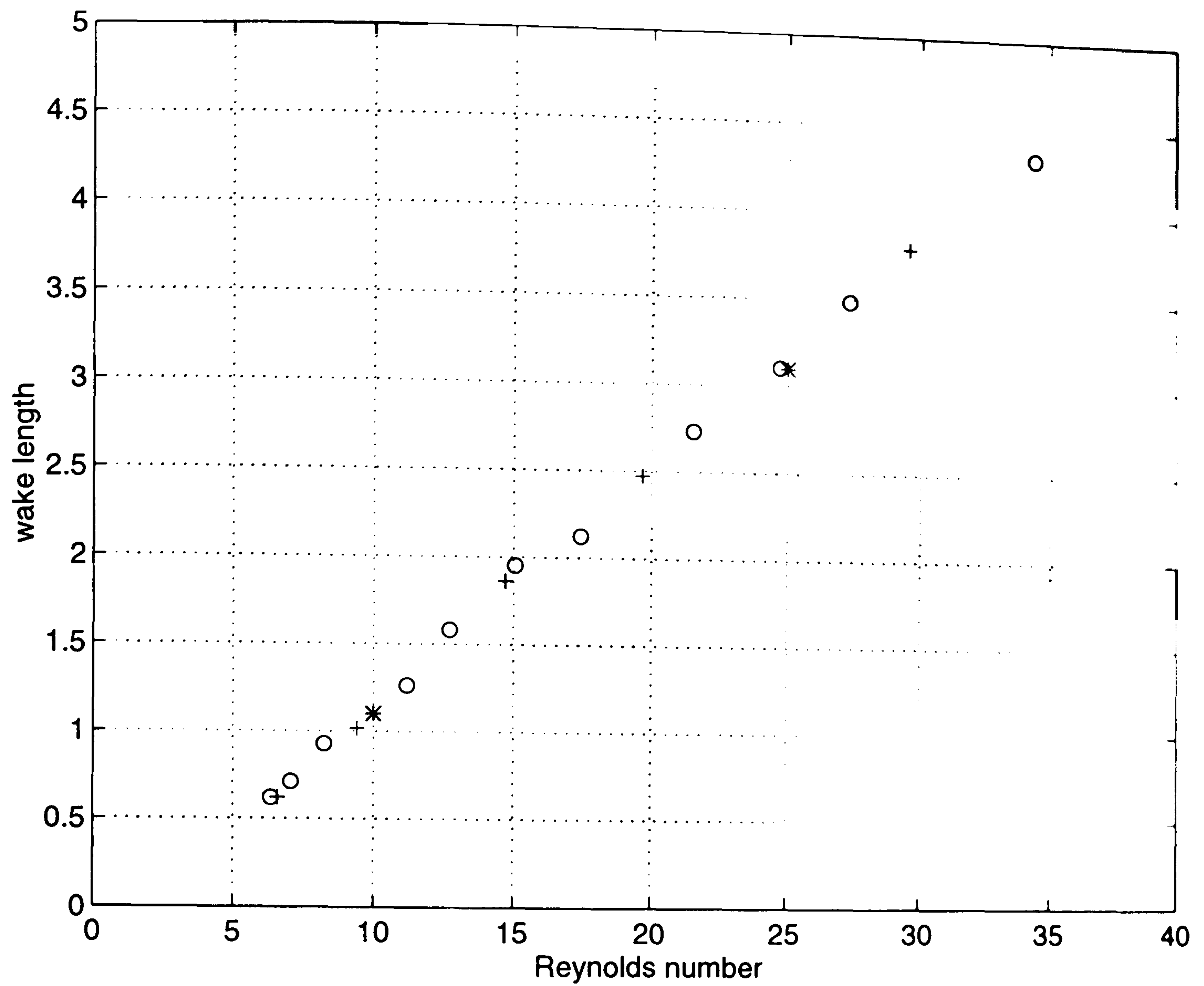


Figure 60: Wake length behind cylinder flow at varying Reynolds numbers. The circles and plus signs represent experimental observations and numerical calculations respectively, as reported by Dennis and Chang [85]. The stars are the values predicted from calculated flows using Shaw's algorithm [19],[20]. This is discussed in Section 8.4.

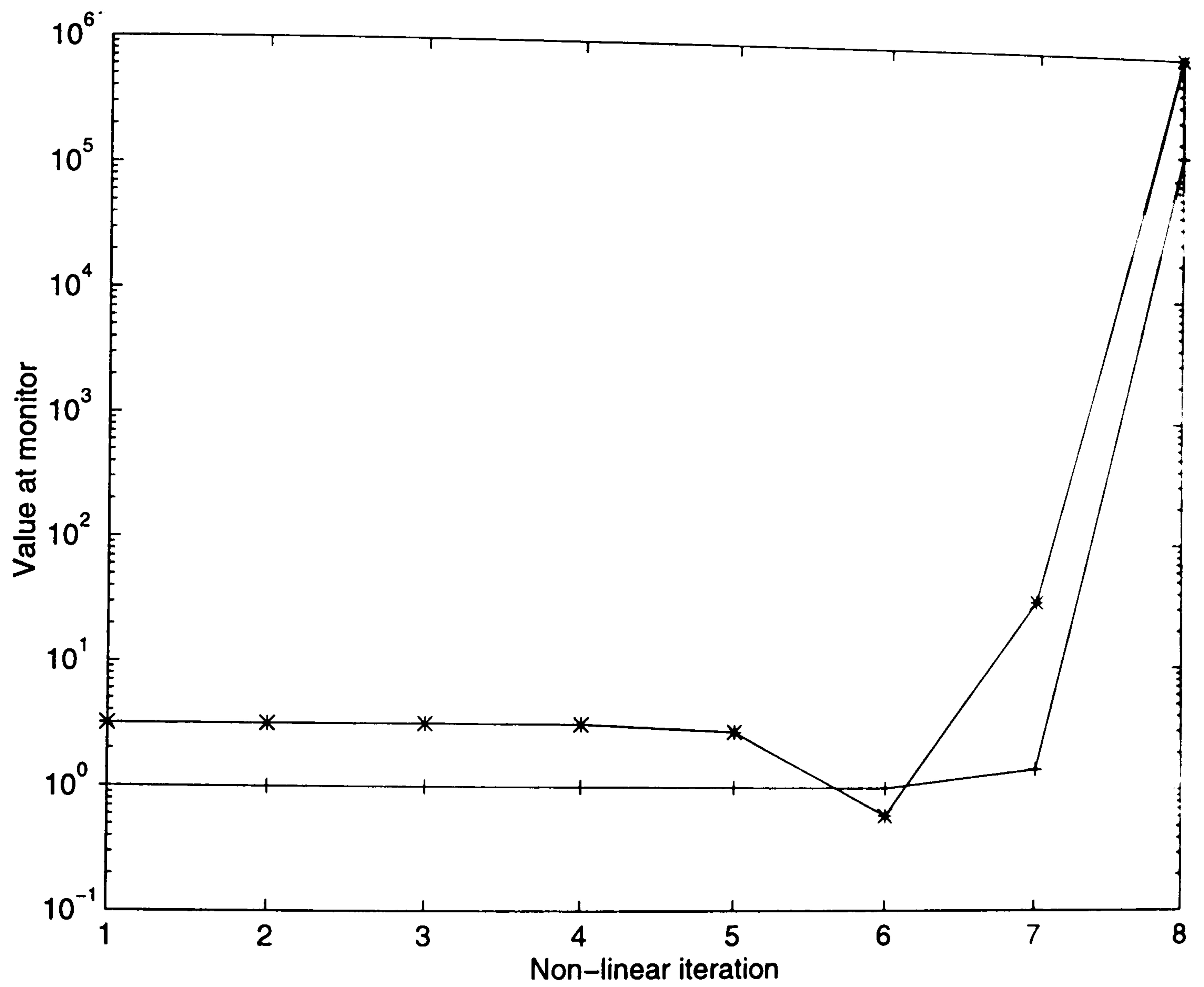


Figure 61: Divergence of monitor values for cylinder flow at $Re=50$. The plus signs represent the velocity at the monitor whilst the star signs represent the pressure. **This** is discussed in Section 8.4.

9 CONCLUSIONS

The goal of the present work was to explore and define strategies for producing fast solutions of the incompressible Navier-Stokes equations. As a starting point, the discretisation method selected was the FE method because of its ability to deal with complex geometries. The formulation used was that of Shaw [19], [20]. This segregated formulation is attractive because the size of the systems of linear equations that must be solved is smaller than the systems of linear equations that would be generated from a coupled formulation. This reduces processing time. Moreover, the formulation is equal-order so that only one mesh needs to be built as opposed to one for pressure and one for velocity.

Whilst an attractive methodology for reducing computation time is to reduce the arithmetic complexity of expressions, this is only really effective on small problems. Whilst problems of 2000 degrees of freedom may then be solved with a 22% reduction in computation time, this reduction in processing time drops markedly as the size of the problem increases, with solution time being dominated by the calculation of the solutions to linear equation systems.

However, the SIMPLE approach used requires the solution of seven linear systems in each non-linear iteration. The use of mass lumping diagonalises the element matrices in the correction stage of the algorithm which imposes the divergence-free condition onto the velocity field, thereby reducing the number of linear equation systems to be solved to just four. This has little effect on the accuracy of the solution scheme. Mass lumping is therefore a cheap way of reducing computation time.

Nevertheless, solution time is still large for problems with many degrees of freedom, so an alternative method of solving linear systems of equations has been explored. Iterative methods, which include line relaxation methods, conjugate gradient type methods and minimum residual methods have been considered. Of these, the conjugate gradient methods have been most useful because they require no parameters to be set, do not require previous solutions to be stored and most importantly have a fast convergence

rate for the matrices produced.

Moreover, the use of a segregated formulation allows exploitation of the properties of the matrices generated, and it is possible to use a symmetric linear equation system solver for the pressure generated matrices, which is more efficient than the more general asymmetric conjugate gradient methods, since only one matrix vector product must be evaluated at each iteration as opposed to two.

Classical primitive variable FE methods are not able to take advantage of this, and moreover require pivoting to handle the zero elements that appear on the leading diagonal by normal discretisation methods.

In a further attempt to reduce computation time, parallel architectures have been considered. Of the architectures available, a SIMD architecture is most attractive because of the repetition of computations across large arrays, even though obtaining high efficiencies on these architectures is known to be difficult. However, straightforward parallelisation of an FE algorithm is complicated by sequential tasks, such as the imposition of boundary conditions. Moreover, the communication that results during normal execution of the code has a detrimental effect on solution time. In dealing with these problems, an approach to the understanding of the data structures and their interaction has been defined. We have considered and defined the concept of data levels and by selection of an appropriate elementary object have been able to construct an implementation in which all operations in the algorithm are performed in parallel (with the exception of reading and writing data). This results in a loss in readability but is necessary to keep communication costs down.

Even with this implementation, the profiles of the parallel solver over various problem sizes indicate that the solution of the linear equation systems still dominate the computation time. This is no surprise considering the number of operations that must be performed, but a large percentage of this time is still occupied by communication.

During the course of this work, a MasPar MP1104 was used. The maximum processing

speed measured on this machine is 145MFlops (in double precision) using 4096 processors which can each store 16Kbytes of memory. The parallel implementation was able to run at an efficiency of 19.2% representing a speedup of 785.0 compared to a Sparc Station 10. This gave a reduction in processing time of over 50%, even though a Sparc Station 10 is a fast computer compared to other sequential computers, whilst the MP1104 is a relatively slow parallel computer. On a fully configured MP2216 which has 16384 processor elements and 64Kbytes of local memory per processor, it is estimated that problems with over 300,000 degrees of freedom could be solved at over 1200 MFlops. This is almost ninety times faster than the processing speed of a Sparc Station 10.

An analysis of moderate Reynolds number flows that would not be possible on a sequential architectures has been undertaken. These flows required special treatment of the convection term. The utility of the upwinding technique has been demonstrated by the flow around a cylinder at a Reynolds number of 25.

Flows at even higher Reynolds numbers however could not be obtained. This suggests that the equal-order formulation which is known to violate the Babuska-Brezzi conditions may be the cause of divergence. At low Reynolds numbers, the flow solutions generated show that this is not problematic. However, at higher Reynolds numbers it is probable that these conditions become an issue.

However, the formulation does not necessarily have to be equal-order. The advantages of an equal-order formulation in terms of reduced time for mesh generation become pointless if a solution cannot be obtained. Moreover, the strategy for parallelisation would allow the implementation of a mixed model formulation on to SIMD architectures. Again, the implementation would need to consider the data levels involved, and since there is effectively one extra mesh, a further nodal and elemental data level must be included in the analysis. Minimisation of the data level interactions via selection of appropriate elementary objects and data level projections would then lead to a successful parallel implementation.

References

- [1] Acheson D.J. *Elementary Fluid Dynamics*. Clarendon Press, Oxford, 1992.
- [2] Tritton D.J. *Physical Fluid Dynamics*. Clarendon Press, Oxford, 2 edition, 1988.
- [3] Roberson J.A. and Crowe C.T. *Engineering Fluid Mechanics*. Houghton Mifflin, Boston, 1993.
- [4] Newton I. *Principia II*. 1687.
- [5] Bernouilli J. *Hydraulica*. 1743.
- [6] Hirsch C. *Numerical Computation of Internal and External Flows: Fundamentals of Numerical Discretisation*, volume 1. John Wiley and Sons Ltd, New York, 1988.
- [7] Canuto C., Hussaini M.Y., Quarteroni A. and Zang T.A. *Spectral Methods in Fluid Dynamics*. Springer-Verlag, New York, 1988.
- [8] Harlow F.H. and Fromm J.E. Computer experiments in fluid dynamics. *Sci. Amer.*, 212:104–110, 1965.
- [9] Patankar S.V. and Spalding D.B. A calculation procedure for heat, mass and momentum transfer in three dimensional parabolic problems. *Int. j. numer. methods heat fluid flow*, 15:1787–1806, 1972.
- [10] Schellbach W. Probleme der Variationsrechnung. *J. Reine Angew. Math.*, 41:293–363. 1851.
- [11] Turner M., Clogh D., Martin H. and Topp L. Stiffness and deflection analysis of complex structures. *J. Aeronaut. Sci.*, 23:805–823. 1956.
- [12] Clough R.W. The finite element method in plane stress analysis. In *Proc. 2nd ASCE Conf. on Electronic Computation*, pages 345–78, 1960.
- [13] Hoff N.J. *Analysis of Structures*. John Wiley and Sons Ltd, New York, 1956.

- [14] Hirsch C. *Numerical Computation of Internal and External Flows: Computational Methods for Inviscid and Viscous Flows*, volume 2. John Wiley and Sons Ltd. New York, 1990.
- [15] Bradshaw P., Cebeci T. and Whitelaw J. *Engineering Calculation Methods for Turbulent Flows*. Academic Press, 1981.
- [16] Glowinski R. and Pironneau O. Finite element methods for Navier-Stokes equations. *Ann. Rev. Fluid Mech.*, 24:167–204, 1992.
- [17] Quinn M.J. *Parallel Computing*. McGraw-Hill, 1994.
- [18] Flynn M.J. Some Computer Organisations and their Effectiveness. *IEEETC*, 9:880–886, 1972.
- [19] Shaw C.T. Adding the time-dependent terms to a segregated finite element solution of the incompressible Navier-Stokes equations. *Engineering Computations*, 8:305–316, 1991.
- [20] Shaw C.T. Using a segregated finite element scheme to solve the incompressible Navier-Stokes equations. *Int. j. numer. methods fluids*, 12:81–92, 1991.
- [21] Press W.H., Teukolsky S.A., Vetterling W.T. and Flannery B.P. *Numerical Recipes in FORTRAN*. Cambridge University Press, Cambridge, 1992.
- [22] Zienkiewicz O.C. and Taylor R.L. *The Finite Element Method: Basic Formulation and Linear Problems*, volume 1. McGraw-Hill, New York, 1989.
- [23] Reddy J.N. *An Introduction to the Finite Element Method*. McGraw-Hill, New York, 1985.
- [24] Patankar S.V. *Numerical Heat Transfer and Fluid Flow*. Hemisphere, New York, 1980.
- [25] Gosman A.D., Pun W.M., Runchal A.K., Spalding D.B., and Wolfshtein M. *Heat and Mass Transfer in Recirculating Flows*. Academic Press, London, 1969.

- [26] Versteeg H.K. and Malalasekera W. *An Introduction to Computational Fluid Dynamics: The Finite Volume Method*. Longman Scientific and Technical, Harlow, England, 1995.
- [27] Babuska I. and Suri M. The p- and h-p versions of the finite element method: an overview. *Comput. Methods Appl. Mech. Eng.*, 80:5–26, 1990.
- [28] Shaw C.T. *Using Computational Fluid Dynamics*. Prentice-Hall, New York, 1992.
- [29] Courant R., Isaacson E. and Reeves M. On the solution of non-linear hyperbolic differential equations by finite differences. *Commun. Pure Appl. Math.*, 5:243–255, 1952.
- [30] Brooks A.N. and Hughes T.J.R. Streamline upwind/Petrov-Galerkin formulation for convection dominated flows with particular emphasis on the incompressible Navier-Stokes equation. *Comput. Methods Appl. Mech. Eng.*, 32:199–259, 1982.
- [31] Zienkiewicz O.C. and Taylor R.L. *The Finite Element Method: Solid and Fluid Mechanics, Dynamics and Non-linearity*, volume 2. McGraw-Hill, New York, 4 edition, 1991.
- [32] Harlow F.H. and Welch J.E. Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface. *Phys. Fluids*, 8:2182–9, 1965.
- [33] Chorin A.J. A numerical method for solving incompressible viscous flow problems. *J. Comput. Phys.*, 2:12–26, 1967.
- [34] Ramaswamy B. and Jue T.C. Segregated finite element formulation of Navier-Stokes equations under laminar conditions. *Fin. Elem. in Analysis and Design*, 9:257–270, 1991.
- [35] Haroutunian V., Engelman M.S. and Hasbani I. Segregated finite element algorithms for the numerical solution of large scale incompressible flow problems. *Int. j. numer. methods fluids*, 17(4):323–348, 1993.

- [36] Pearson C.E. A Computational method for viscous flow problems. *J. Fluid Mech.*, 21:611–635, 1965.
- [37] Campion-Renson A. and Crochet M.J. On the streamfunction vorticity finite element solutions of Navier-Stokes equations. *Int. j. numer. methods eng.*, 12:1809–1818, 1978.
- [38] Dhatt G., Fomo B.K. and Bourque C.A. A streamfunction-vorticity finite element formulation for the Navier-Stokes equations. *Int. j. numer. methods eng.*, 17:199–212, 1981.
- [39] Peeters M.F., Habashi W.G. and Dueck E.G. Finite element stream function-vorticity solutions of the incompressible Navier-Stokes equations. *Int. j. numer. methods fluids*, 7:17–27, 1987.
- [40] Guevremont G., Habashi W.G. and Hafez M.M. Finite element solution of the Navier-Stokes equations by a velocity-vorticity method. *Int. j. numer. methods fluids*, 10:461–475, 1990.
- [41] Chorin A.J. Numerical solution of the Navier-Stokes equations. *Math. Comput.*, 22:745–762, 1968.
- [42] Haroutunian V., Engelman M. and Hasbani I. Three segregated finite element solution algorithms for the numerical solution of incompressible flow problems. *Advances in Finite Element Analysis in Fluid Dynamics American Society of Mechanical Engineers*, 1991.
- [43] Chao Y.C. and Ho W.C. Behaviour of five solution algorithms on turbulent calculations. *Comm. Appl. Numer. Methods*, 5:253–261, 1989.
- [44] Tamamidis P. and Assanis D.N. Prediction of three dimensional viscous flows using body-fitted coordinates. *Multidisciplinary Applications of Computational Fluid Dynamics (ASME Fluids Engineering Division)*, 129:99–107, 1991.
- [45] Issa R.I. Solution of the implicit discretised fluid flow equations by operator splitting. *J. Comput. Phys.*, 62:40–65, 1986.

- [46] Jang D.S., Jetli R. and Acharya S. Comparison of the PISO, SIMPLER and SIM-
PLEC algorithms for the treatment of the pressure-velocity coupling in steady flow
problems. *Numer. Heat Transfer*, 10:209–228, 1986.
- [47] Ahmadi-Befrui B., Gosman A.D., Issa R.I. and Watkins A.P. An implicit non-
iterative solution procedure for the calculation of flows in reciprocating engine cham-
bers. *Comput. Methods Appl. Mech. Eng.*, 79:249–279, 1990.
- [48] Benim A.C. and Zinser W. A segregated formulation of the Navier-Stokes equations
with finite element. *Comput. Methods Appl. Mech. Eng.*, 57:223–237, 1986.
- [49] Rice J.G. and Schnipke R.J. An equal-order velocity-pressure formulation that does
not exhibit spurious pressure modes. *Comput. Methods Appl. Mech. Eng.*, 58:135
149, 1986.
- [50] Dhatt G. and Touzot G. *The Finite Element Method Displayed*. John Wiley and
Sons Ltd, New York, 1984.
- [51] Griffiths D.V. and Smith I.M. *Numerical Methods for Engineers*. Blackwell Scientific
Publications, London, 1991.
- [52] Gresho P.M., Chan S.T. Lee R.L. and Upson C.D. A modified finite element method
for solving the time-dependent incompressible Navier-Stokes equations. Part 1: The-
ory. *Int. j. numer. methods fluids*, 4:557–598, 1984.
- [53] Fletcher C.A.J. *Computational Techniques for Fluid Dynamics, 1: Fundamental
and general techniques*. Springer-Verlag, Berlin, 1991.
- [54] Young D.M. *Iterative Solution of Large Linear Systems*. Academic, New York, 1971.
- [55] Howard D., Connelly W.M. and Rollett J.S. Unsymmetric conjugate gradient meth-
ods and sparse direct methods in finite element flow simulation. *Int. j. numer.
methods fluids*, 10:925–945, 1990.
- [56] Saad Y. and Schultz M.H. GMRES: A generalised minimal residual algorithm for
solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7(3), 1986.

- [57] Habashi W.G., Robichaud M., Nguyen V.N., Ghaly W.S., Fortin M., and Liu J.W.H. Large scale CFD by the finite element method. *Int. j. numer. methods fluids*, 18:1083–1105, 1994.
- [58] Chin P., D’Azevedo E.F., Forsyth P.A. and Tang W.P. Preconditioned conjugate gradient methods for the incompressible Navier-Stokes equations. *Int. j. numer. methods fluids*, 15:273–295, 1992.
- [59] Turner P.R., editor. *Lecture notes in mathematics: Numerical Analysis and Parallel Processing*, volume 1397. Springer-Verlag, Berlin, 1989.
- [60] Mallick S. and Shaw C.T. Segregated finite element algorithms on massively parallel machines. In Morgan K., Onate E., Periaux J., Peraire J. and Zienkiewicz O.C., editor, *Proceedings of VIIIth Int. Conf. on Finite Elements in Fluids*, pages 1231–1240, Barcelona, Spain, Sept 20-24 1993. Pineridge Press.
- [61] Fischer P.F. and Patera A.T. Parallel simulation of viscous incompressible flows. *Ann. Rev. Fluid Mech.*, 24:483–527, 1994.
- [62] Brauml T. *Parallel Programming: An Introduction*. Prentice-Hall, 1993.
- [63] Hockney R.W. and Jesshope C.R. *Parallel Computers 2 - Architecture, Programming and Algorithms*. Institute of Physics Publishing, Bristol, 1988.
- [64] Oden J.T. and Patra A. A parallel adaptive strategy for h-p finite element computations. *Comput. Methods Appl. Mech. Eng.*, 121:449–470, 1995.
- [65] Sawley M.L. Control and data parallel methodologies for flow calculations. In *Supercomputing Europe*, Utrecht, February 1993.
- [66] Barragy E. and Van de Geijn R. Performance and scalability of finite element analysis for distributed parallel computation. *J. Parallel Distrib. Comput.*, 21:202–212, 1994.
- [67] Natarajan R. and Pattnaik P. Performance of the conjugate gradient method on VICTOR. *J. Comput. Phys.*, 100:396–401, 1992.

- [68] Sawley M.L. and Tegner J.K. A data parallel approach to multiblock flow computations. *Int. j. numer. methods fluids*, 19:707–721, 1994.
- [69] Kennedy J.G., Behr M., Kalro V. and Tezduyar T.E. Implementation of implicit finite element methods for incompressible flows on the CM-5. *Comput. Methods Appl. Mech. Eng.*, 119(1–2):95–111, 1994.
- [70] Johnsson S.L. and Mathur K.K. Data structures and algorithms for the finite element method on a data parallel supercomputer. *Int. j. numer. methods eng.*, 29(4):881–908, 1990.
- [71] Liou J. and Tezduyar T.E. Clustered Element-by-Element Computations for Fluid Flow. In Horst S.D., editor, *Parallel Computational Fluid Dynamics: Implementation and Results*, chapter 9. MIT Press, London, England, 1989.
- [72] Hughes T.J.R., Levit I. and Winget J. An element-by-element solution algorithm for problems of structural and solid mechanics. *Comput. Methods Appl. Mech. Eng.*, 36:241–254, 1983.
- [73] Lai C.H. and Liddell H.M. Preconditioned conjugate gradient methods on the DAP. In *Proceedings from The Mathematics of Finite Elements and Applications VI*, pages 145–156. Academic Press, 1988.
- [74] Jacobsen K.P. Data mapping strategies for efficient implementation of a CFD program on a massively parallel computer. In *Proceedings of 4th Int. Symp. of CFD*, Sept 9-12 1991.
- [75] Lang B. A parallel algorithm for reducing symmetrical banded matrices to tridiagonal form. *SIAM J. Sci. Comput.*, 14(6):1320–1338, 1993.
- [76] Sawley M.L. and Bergman C.M. A Comparative study of the use of the data parallel approach for compressible flow calculations. *Par. Comp.*, 20(3):363–373, 1994.
- [77] Tezduyar T.E., Aliabadi S., Behr M., Johnson A. and Mittal S. Massively parallel finite element computation of three dimensional flow problems. In *Proceedings of 6th Japan Numer. Fl. Dynamics. Symp.*, 1992.

- [78] Kelly D.W., Nakazawa S. and Zienkiewicz O.C. A note on anisotropic balancing dissipation in the finite element method approximation to convective diffusion problems. *Int. j. numer. methods eng.*, 15:1705–1711, 1980.
- [79] Armaly B.F., Durst F., Pereira J.C.F. and Schonung B. Experimental and theoretical investigation of backward-facing step flow. *J. Fluid Mech.*, 127:473–496, 1983.
- [80] Gartling D.K. A test problem for outflow boundary conditions: flow over a backward facing step. *Int. j. numer. methods fluids*, 11:953–967, 1990.
- [81] Gresho P.M., Gartling D.K., Torczynski J.R., Cliffe K.A., Winters K.H., Garratt T.J., Spence A. and Goodrich J.W. Is the steady viscous incompressible 2-dimensional flow over a backward-facing step at $Re=800$ stable? *Int. j. numer. methods fluids*, 17:501–541, 1993.
- [82] Fornberg B. A numerical study of steady viscous flow past a circular cylinder. *J. Fluid Mech.*, 98:819–855, 1980.
- [83] Panton R.L. *Incompressible Flow*. John Wiley and Sons Ltd, New York, 1996.
- [84] Dennis S.C.R. and Chang G. Numerical solutions for steady flow past a circular cylinder at Reynolds numbers up to 100. *J. Fluid Mech.*, 42:471–489, 1970.
- [85] Burggraf O.R. Analytical and numerical studies of the structure of steady separated flows. *J. Fluid Mech.*, 24:113–151, 1966.
- [86] Engelman M.S. and Jamnia M-A. Transient flow past a circular cylinder: a benchmark solution. *Int. j. numer. methods fluids*, 11:985–1000, 1990.
- [87] MasPar Computer Corporation. *MasPar System Overview*, 1992.
- [88] MasPar Computer Corporation. *MasPar Fortran User Guide*, 1992.

A Iterative linear equation solvers

In the following sections, we assume that a matrix equation needs to be solved, which has the form of equation (96).

$$Ax = b \quad (96)$$

A.1 Line relaxation methods

The Jacobi, Gauss-Seidel and SOR methods are classed as line relaxation methods.

Jacobi Method

$$x_i^{(k+1)} = (b_i - \sum_{j=1}^n a_{ij}x_j^{(k)})/a_{ii} \quad (97)$$

Gauss Seidel Method

$$x_i^{(k+1)} = (b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^n a_{ij}x_j^{(k)})/a_{ii} \quad (98)$$

Successive Over-Relaxation (SOR)

$$x_i^{(k+1)} = \frac{\omega}{a_{ii}} \left[b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^n a_{ij}x_j^{(k)} \right] + (1 - \omega)x_i^{(n)} \quad (99)$$

For convergence: $0 < \omega < 2$

A.2 The conjugate gradient (CG) method and its derivative

CG methods are direct methods if exact arithmetic could be used, but are classed as iterative methods since reasonable solutions can be obtained in far fewer steps than the theoretical maximum number [55].

The conjugate gradient method is a popular method for solving large linear systems. It is easy to program and use, and is ideal for parallelisation. It is derived from the steepest descent method.

The gradient (or conjugate direction) techniques look at solving equation (96) by a functional approach. If we define a function $\Theta(x)$ by:

$$\Theta(x) = \frac{1}{2}x^T Ax - x^T b \quad (100)$$

then the minimum value of $\Theta(x)$ equals $-\frac{1}{2}b^T A^{-1}b$, which corresponds to setting $x = A^{-1}b$. Thus minimising $\Theta(x)$ and solving $Ax = b$ are equivalent problems.

Algorithm: Method of conjugate gradients (CG) [55]

$$r_0 = p_0 = b - Ax_0$$

Repeat until convergence:

$$\alpha_n = \langle r_n, p_n \rangle / \langle p_n, Ap_n \rangle$$

$$x_{n+1} = x_n + \alpha_n p_n$$

$$r_{n+1} = r_n - \alpha_n Ap_n$$

$$\beta_n = - \langle r_{n+1}, Ap_n \rangle / \langle p_n, Ap_n \rangle$$

$$p_{n+1} = r_{n+1} + \beta_n p_n$$

alternative formulae for α and β , using orthogonality and conjugacy relations:

$$\alpha_n = \langle r_n, r_n \rangle / \langle p_n, Ap_n \rangle$$

$$\beta_n = \langle r_{n+1}, r_{n+1} \rangle / \langle r_n, r_n \rangle$$

Algorithm: Conjugate gradient squared (CGS) method

$$R_0 = P_0 = K_0 = b - AX_0.$$

Repeat until convergence:

$$\alpha_n = r_o^T K_n / r_o^T AP_n$$

$$H_n = K_n - \alpha_n AP_n$$

$$R_{n+1} = R_n - \alpha_n A(H_n + K_n)$$

$$X_{n+1} = X_n + \alpha_n (H_n + K_n)$$

$$\beta_n = \mathbf{r}_0^T \mathbf{R}_{n+1} / \mathbf{r}_0^T \mathbf{R}_n$$

$$\mathbf{P}_{n+1} = \mathbf{R}_{n+1} + 2\beta_n \mathbf{H}_n + \beta^2 \mathbf{P}_n$$

$$\mathbf{K}_{n+1} = \mathbf{R}_{n+1} + \beta_n \mathbf{H}_n$$

Algorithm: Biconjugate Gradient (BICG)

$$\mathbf{r}_0 = \mathbf{p}_0 = \bar{\mathbf{r}}_0 = \bar{\mathbf{p}}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$$

Repeat until convergence:

$$\alpha_n = \langle \bar{\mathbf{p}}_n, \mathbf{r}_n \rangle / \langle \bar{\mathbf{p}}_n, \mathbf{A}\mathbf{p}_n \rangle$$

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \alpha_n \mathbf{p}_n$$

$$\mathbf{r}_{n+1} = \mathbf{r}_n - \alpha_n \mathbf{A}\mathbf{p}_n$$

$$\bar{\mathbf{r}}_{n+1} = \bar{\mathbf{r}}_n - \alpha_n \mathbf{A}^T \bar{\mathbf{p}}_n$$

$$\beta_n = \langle \bar{\mathbf{r}}_{n+1}, \mathbf{A}\mathbf{p}_n \rangle / \langle \bar{\mathbf{p}}_n, \mathbf{A}\mathbf{p}_n \rangle$$

$$\mathbf{p}_{n+1} = \mathbf{r}_{n+1} + \beta_n \mathbf{r}_n$$

$$\bar{\mathbf{p}}_{n+1} = \bar{\mathbf{r}}_{n+1} + \beta_n \bar{\mathbf{r}}_n$$

This is valid for unsymmetric matrices and reduces to the CG in the symmetric case.

A.3 Minimum residual methods

The minimum residual methods include ORTHOMIN and the GMRES algorithm. A full discussion of these is given by Saad and Schultz [56], Howard *et al* [55] and Habashi *et al* [57].

B Manchester's MasPar machine

The MasPar (MP1104) is of SIMD architecture, and so potential problems of scheduling processors that might occur on MIMD architectures are not a consideration. Moreover SIMD machines are scalable and designs for this particular architecture allow it to have up to 16384 processors. Other SIMD architectures have been designed, scalable up to 65536 processors. General programming principles from one SIMD machine to another are similar (in a high-level language sense), thus, whilst languages differ syntactically from one platform to the next, methodologies which may be developed on the MasPar are applicable to machines of similar architecture. Different platforms do differ in the low-level constructs that may be used to reduce or optimise certain procedures. For example, it is possible to hard-wire communication routes into a program during the compilation process on the Connection Machine-200, thereby reducing startup time for communication. Moreover, an increase in the number of processors, can be automatically exploited with judicious programming.

This is one of the main advantages of distributed memory-SIMD platforms: application can instantly reap the benefits of scalability, should the number of processors on the machine be increased (or for that matter, if the power of each processor is increased).

B.1 Specifications

The MasPar MP1104 is a SIMD massively parallel computer, and each machine consists of a front-end workstation (FE) which acts as a host to a Data Parallel Unit (DPU) [87]. The FE is a standard DECstation 5000 Model 200, which runs ULTRIX (Digital version of UNIX). Various numbers of processors are available on the DPU ranging from 1024 to 16384, but the machine used has 4096 processor elements (PEs) arranged in a 64x64 grid and is theoretically capable of 145 MFlops when using double precision arithmetic. Each PE is a custom-made 1.8 MIPS RISC-like processor, connected to 64 Kbytes of local memory. This gives a total of 64 Mbytes for the DPU. Object code

resides partly on the FE and partly on the DPU. When executing a program, microcode for the parallel subsections is processed on the DPU, whilst the program runs as a UNIX process on the FE. Data flows to and from the FE and DPU as the program executes via a VME-databus (a Digital product).

B.2 Data transfer and inter-processor communication

Besides data transfer between the FE and the DPU (which is the slowest form of communication), the MasPar allows communication between PEs. Two forms exist: an arbitrary PE can communicate with any other using the global router; this is the slowest form of PE-PE communication, with a peak throughput of 1.3 Gbyte/s. X-net is a much faster method of communication, which allows each PE to exchange data with its nearest neighbour in any vertical, horizontal or diagonal direction, with a throughput of 1.3 Gbyte/s. Whilst inter-processor communication (IPC) is transparent to the programmer, the way in which a problem is formulated can significantly affect the run time.

B.3 Programming Tools

Software is provided to enable the user to control the linking of the FE and DPU effectively [87]. The MasPar Programming Environment (MPPE) enables programs to be debugged and optimised, and two programming languages are presently available, the MasPar Application Language (MPL) and MasPar FORTRAN (or MPFORTRAN) [8]. MPL is a low-level compiled language based on C, which allows programming of the DPU directly, whereas MasPar FORTRAN is based on the FORTRAN 77 ANSI standard with parallel data array extensions from the ANSI FORTRAN 8x proposal. The major difference between the two languages is that with MPL, you can explicitly control how data is mapped to the processors, and select the communication method that optimises the speed of operation. With MPFORTRAN, all the virtualisation and data movement is handled implicitly, though it is highly dependent on the way in which the program

FORTRAN-77	MPFORTRAN
dimension x(4096)	real x(4096)
do 10 i=1,4096	x=0.0
x(i)=0.0	
10 continue	

Table 15: Simple comparison of FORTRAN-77 and MPFORTRAN

written.

Moreover, MPL does not conform to any particular standard for machines of SIMD type and so it has not been used in order to allow us to formulate general strategies for SIMD machines. As will be shown, this is not as restrictive as it may appear to be, and so allows reasonable implementations to be produced.

As an example of the use of the data parallel features, let us consider the problem of setting a one-dimensional array to zero. The two portions of coding below, in Table 15, show how this is done for both FORTRAN 77 and MPFORTRAN.

On the left, FORTRAN 77 uses do-loop constructions whereas MPFORTRAN treats the array as a single entity. Here, each PE is allocated the job of setting one of the elements of the array to zero, taking the same amount of time as setting one scalar variable to zero. Hence the parallel process is speeded up 4096 times.

B.4 Array Mapping

Arrays can be stored on either the FE or the DPU, controlled by compiler directives. Referencing a DPU array with FORTRAN 77 syntax results in the array being passed to the FE for manipulation, as the calculation is seen as being a scalar process. Similarly, referencing a FE array with vector syntax results in the array being passed to the DPU.

Since both of these situations involve the slowest form of data transfer, i.e. between the FE and the DPU, they are to be avoided. The phenomenon whereby large amounts of data are swapped between the DPU and FE is known as *sloshing* and is one of the main causes of poor efficiency. Hence we can see that the storage location of arrays, either on the DPU or FE, is an important problem in translating a FORTRAN 77 code for sequential machines into MPFORTRAN for the MasPar. To prevent sloshing, essential sequential processes (such as the imposition of boundary conditions) must be effectively handled so that these processes do not slow down the implementation. This precludes the possibility of using the FE for processing array variables that have been generated on the DPU, unless parallelisation of the process involved is highly impractical. In this situation it is usually best to find an alternative method for evaluating the desired values.

Another source of inefficiency stems from the way in which arrays are mapped onto PEs themselves, as IPC must be kept as small as possible. In the worst possible case, for example, algebraic calculations of two arrays mapped poorly may take 20 times longer than arrays mapped so that the arithmetic is performed using no IPC, simply due to the time for global router communication.

C Mapping Variables on the MasPar

The data structure for the elemental data is then defined in the following way. Since the same algebraic manipulations are performed for each element, an obvious choice for the structure is to map one element to one VP, and to keep as much data as possible pertaining to that element there. For example, using a trilinear hexahedral element (without loss of generality) produces an 8x8 element matrix for each element. Thus we can define arrays *elhs* and *erhs* which contain all the left hand side and right hand side element matrix data in the following manner:

```
real, dimension (npelem, 8, 8) :: elhs
```

```
CMPF MAP elhs(allbits, memory, memory)
```

```
real, dimension (npelem, 8) :: erhs
```

```
CMPF MAP erhs(allbits, memory)
```

where npelem is the number of elements in the mesh

The CMPF MAP commands are compiler directives which tell the DPU to map the first dimension of the array over the VPs, and to map the other dimensions into corresponding local memory.

The nodal data is exclusively one-dimensional, so mapping each node to a processor can be done simply by:

```
real, dimension (npnode) :: uold
```

where npnode is the number of nodes in the problem.

One dimensional arrays are automatically mapped across all the processors (unless otherwise directed by CMPF MAP command). This will facilitate algebraic manipulation relating to the same node and fortunately the exclusively nodal data calculations this form (i.e. predicting the new flow variables).